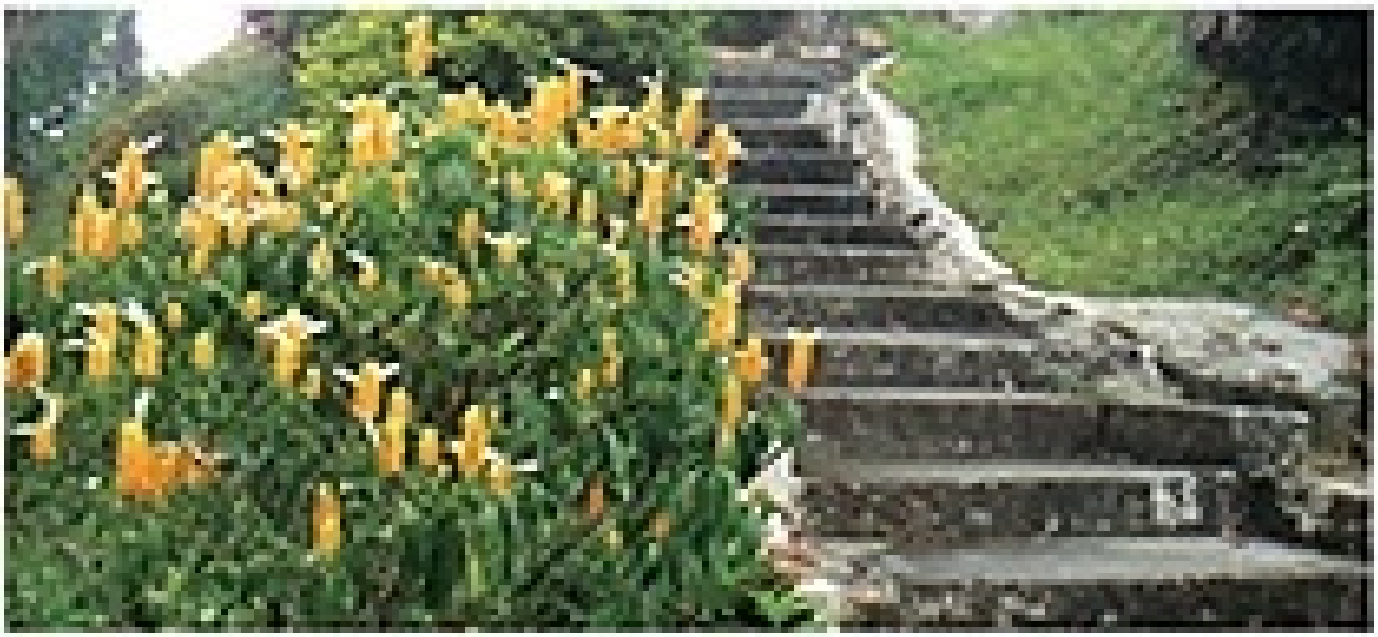
Профессиональное программирование

Scala

3-е издание



Scala 2.12. Впервые на русском языке

Мартин Одерски
Лекс Спун
Билл Веннерс

Питер
artima

Профессиональное программирование

Scala

3-е издание

Scala 2.12. Впервые на русском языке

Мартин Одерски
Лекс Спун
Билл Веннерс

ПИТЕР
artima

Мартин Одерски, Лекс Спун, Билл Веннерс

Scala. Профессиональное программирование

ПИТЕР®

2017

Переводчики *Н. Вильчинский, С. Черников, Ю. Яковлева*
Технические редакторы *Н. Гринчик, Н. Рощина*
Литературный редактор *Н. Рощина*
Художники *С. Заматевская , Г. Синякина (Маклакова)*
Корректоры *Т. Курьянович, Т. Радецкая*
Верстка *О. Богданович*

Мартин Одерски, Лекс Спун, Билл Веннерс
Scala. Профессиональное программирование . — СПб.: Питер, 2017.

ISBN 978-5-496-02951-3
© ООО Издательство "Питер", 2017

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Предисловие

Вы очень вовремя приобрели эту книгу! Язык Scala становится все более популярным, количество его приверженцев неизменно увеличивается, а объявлений о вакансиях программистов на Scala все больше и больше. Неважно, по каким именно причинам вы занимаетесь программированием — увлечены им или пытаетесь на этом заработать (или вами движет и то и другое), — перед получаемым удовольствием и повышением производительности труда при работе с языком Scala трудно устоять. Лично мне удовлетворение от работы приносит решение сложных задач простыми и рациональными методами. Цель Scala состоит не только в предоставлении такой возможности, но и в том, чтобы сделать вашу работу приятной, и в этой книге вы узнаете, как воспользоваться всеми преимуществами данного языка.

В своих первых экспериментах я работал с версией Scala 2.5 и сразу оценил присущее этому языку синтаксическое и концептуальное постоянство. Столкнувшись с нестандартной ситуацией, когда параметры типа сами по себе не могли иметь параметры типа, я (испытывая некоторую робость) подошел к Мартину Одерски (Martin Odersky) на конференции, проводившейся в 2006 году, и предложил пройти стажировку в его команде, чтобы попытаться устранить это ограничение. Мой вклад в общее дело был принят, и в версии Scala 2.7 и выше появилась поддержка полиморфизма конструктора типов. В дальнейшем я работал над большим количеством других частей компилятора. В 2012 году я перешел с временной ставки стажера в лаборатории Мартина на должность руководителя команды Typesafe, как только Scala, добравшись до версии 2.10, перерос из научно-образовательного в надежный язык промышленного уровня.

Версия Scala 2.10 стала поворотной точкой: можно отметить переход от постоянно обновляющихся многофункциональных выпусков, ориентированных на научную работу, к упрощению

языка и более широкому внедрению на производстве. Мы переключили свое внимание на вопросы, не описанные в диссертациях, например двоичную совместимость между основными выпусками. Развивая и совершенствуя платформу Scala, мы стараемся обеспечить ее стабильность и работаем над уменьшением объема основной библиотеки, которую стремимся стабилизировать. Чтобы это стало возможным, в первом проекте по разработке Scala, где я был техническим руководителем, мы ввели модульную организацию стандартной библиотеки Scala версии 2.11.

В попытке снизить темпы изменений в Typesafe также было решено чередовать изменения библиотеки и компилятора. В этой книге рассказывается о Scala 2.12, которая характеризуется обновленной версией компилятора, нацеленного на максимальное использование новых возможностей Java 8. Для взаимодействия с Java и получения тех же выгод от оптимизаций JVM-машины в Scala функции преобразуются в тот же байт-код, который вырабатывается компилятором Java 8. По аналогии с этим трейты Scala теперь переводятся в интерфейсы Java с исходными методами. Обе схемы компиляции уменьшают количество ухищрений, к которым приходилось прибегать в прежних компиляторах Scala, ориентируют нас на более тесное применение платформы Java, сокращают время компиляции и одновременно с этим повышают производительность. В придачу получаем более однородную двоичную совместимость!

Эти улучшения платформы Java 8 весьма важны для Scala, и очень приятно видеть, что Java придерживается тех тенденций, которые выстраивались Scala на протяжении более 10 лет! Несомненно, в Scala более широко используется функциональное программирование с применением исходной неизменяемости, единой трактовкой выражений (в этой книге вы вряд ли найдете обратное утверждение), поиском по шаблону, вариантностью по месту объявления (применяемая в Java вариантность по месту использования делает крайне затруднительным функциональное

подтипирование) и т. д.! Иными словами, там больше функционального программирования, чем красивого синтаксиса для лямбда-выражений.

Наша цель как «распорядителей» языка заключается в создании не только его основ, но и экосистемы. Успех Scala достигается благодаря множеству великолепных библиотек, выдающихся интегрированных сред разработки и инструментальных средств, а также благодаря исключительно ценным участникам нашего сообщества. Мне очень понравилось первое десятилетие работы над Scala в качестве реализатора языка, и меня весьма волнует и вдохновляет то, что многие программисты получают удовольствие от использования Scala в различных сферах деятельности.

Мне нравится программировать на Scala, и я надеюсь, что и вам это придется по душе. И от всего сообщества Scala я говорю вам: добро пожаловать!

Адриан Мурс (Adriaan Moors)

Сан-Франциско, Калифорния

14 января 2016 года

Благодарности

Мы благодарны за вклад в эту книгу и в рассматриваемые в ней материалы многим людям.

Сам язык Scala является плодом усилий множества специалистов. Свой вклад в проектирование и реализацию версии 1.0 внесли Филипп Альтер (Philippe Altherr), Винсент Кремет (Vincent Cremet), Жиль Дюбоше (Gilles Dubochet), Бурак Эмир (Burak Emir), Стефан Мишель (Stéphane Micheloud), Николай Михайлов (Nikolay Mihaylov), Мишель Шинц (Michel Schinz), Эрик Стенман (Erik Stenman) и Матиас Зенгер (Matthias Zenger). К разработке второй и текущей версий языка, а также инструментальных средств подключились Фил Багвел (Phil Bagwell), Антонио Куней (Antonio Cunei), Юлиан Драгос (Iulian Dragos), Жиль Дюбоше (Gilles Dubochet), Мигель Гарсиа (Miguel Garcia), Филипп Халлер (Philipp Haller), Шон Макдирмид (Sean McDirmid), Инго Майер (Ingo Maier), Донна Малайери (Donna Malayeri), Адриан Мурс (Adriaan Moors), Хуберт Плоциничак (Hubert Plociniczak), Пол Филипс (Paul Phillips), Александр Прокопец (Aleksandar Prokopec), Тиарк Ромпф (Tiark Rompf), Лукас Рыц (Lukas Rytz) и Джеффри Уошберн (Geoffrey Washburn).

Следует также упомянуть тех, кто участвовал в формировании конструкции языка. Эти люди любезно делились с нами своими идеями в оживленных и вдохновляющих дискуссиях, вносили важные фрагменты кода в ходе работы с открытыми источниками и делали весьма ценные замечания по поводу предыдущих версий данного документа. Это Гилад Браха (Gilad Bracha), Натан Бронсон (Nathan Bronson), Коаюан (Caoyuan), Эймон Кэннон (Aemon Cannon), Крейг Чамберс (Craig Chambers), Крис Конрад (Chris Conrad), Эрик Эрнст (Erik Ernst), Матиас Феллизен (Matthias Felleisen), Марк Харра (Mark Harrah), Шрирам Кришнамурти (Shriram Krishnamurti), Гэри Ливенс (Gary Leavens), Дэвид Макивер (David MacIver), Себастьян Манит (Sebastian Maneth), Рикард

Нильссон (Rickard Nilsson), Эрик Мейер (Erik Meijer), Лалит Пант (Lalit Pant), Дэвид Поллак (David Pollak), Джон Претти (Jon Pretty), Клаус Остерман (Klaus Ostermann), Хорхе Ортис (Jorge Ortiz), Дидье Реми (Didier Rйmy), Майлз Сабин (Miles Sabin), Виджей Сарасват (Vijay Saraswat), Даниэль Спивак (Daniel Spiewak), Джеймс Страчан (James Strachan), Дон Симе (Don Syme), Эрик Торребор (Erik Torreborre), Мэдс Торгерсен (Mads Torgersen), Филипп Уодлер (Philip Wadler), Джейми Уэбб (Jamie Webb), Джон Уильямс (John Williams), Кевин Райт (Kevin Wright) и Джейсон Зауг (Jason Zaugg). Очень полезные отзывы, которые помогли нам улучшить язык и его инструментальные средства, были получены от подписчиков на наши рассылки по Scala.

Джордж Бергер (George Berger) усердно работал над тем, чтобы процесс создания книги и ее размещения в Интернете протекал гладко. В результате в данном проекте не было никаких технических сбоев.

Ценные отзывы о начальных вариантах текста книги были получены нами от многих людей. Наших благодарностей заслуживают Эрик Армстронг (Eric Armstrong), Джордж Бергер (George Berger), Алекс Блевитт (Alex Blewitt), Гилад Браха (Gilad Bracha), Уильям Кук (William Cook), Брюс Экель (Bruce Eckel), Стефан Мишель (Stйphane Micheloud), Тод Мильштейн (Todd Millstein), Дэвид Поллак (David Pollak), Фрэнк Соммерс (Frank Sommers), Филипп Уодлер (Philip Wadler) и Матиас Зенгер (Matthias Zenger). Хотим сказать спасибо представителям Silicon Valley Patterns group за их весьма полезный обзор. Это Дейв Астелс (Dave Astels), Трейси Бялик (Tracy Bialik), Джон Брюер (John Brewer), Эндрю Чейз (Andrew Chase), Брэдфорд Кросс (Bradford Cross), Рауль Дюк (Raoul Duke), Джон П. Эйрих (John P. Eurich), Стивен Ганц (Steven Ganz), Фил Гудвин (Phil Goodwin), Ральф Йочем (Ralph Jocham), Ян-Фа Ли (Yan-Fa Li), Тао Ма (Tao Ma), Джефри Миллер (Jeffery Miller), Суреш Пай (Suresh Pai), Русс Руфер (Russ Rufer), Дэйв У. Смит (Dave W. Smith), Скотт Торнквест (Scott Turnquest), Вальтер Ваннини (Walter Vannini), Дарлин Уоллах (Darlene Wallach) и

Джонатан Эндрю Уолтер (Jonathan Andrew Wolter). Хочется также поблагодарить Дуэйна Джонсона (Dewayne Johnson) и Кима Лиди (Kim Leedy) за помощь в художественном оформлении обложки и Фрэнка Соммерса (Frank Sommers) — за работу над алфавитным указателем.

Хотелось бы также выразить особую благодарность всем нашим читателям, приславшим комментарии. Они нам очень пригодились для повышения качества книги. Мы не в состоянии опубликовать имена всех, приславших комментарии, но объявим имена тех читателей, кто прислал не менее пяти комментариев на стадии eBook PrePrint™, воспользовавшись ссылкой **Suggest**. Отсортируем их имена по убыванию количества комментариев, а затем в алфавитном порядке. Наших благодарностей заслуживают Дэвид Бизак (David Biesack), Дон Стефан (Donn Stephan), Матс Хенриксон (Mats Henricson), Роб Диккенс (Rob Dickens), Блэр Захак (Blair Zajac), Тони Слоан (Tony Sloane), Найджел Харрисон (Nigel Harrison), Хавьер Диас Сото (Javier Diaz Soto), Уильям Хелан (William Heelan), Джастин Фурдер (Justin Forder), Грегор Пёрди (Gregor Purdy), Колин Перкинс (Colin Perkins), Бьярте С. Карлсен (Bjarte S. Karlsen), Эрвин Варга (Ervin Varga), Эрик Уиллигерс (Eric Williger s), Марк Хейс (Mark Hayes), Мартин Элвин (Martin Elwin), Калум Маклин (Calum MacLean), Джонатан Уолтер (Jonathan Wolter), Лес Прушински (Les Pruszynski), Сет Тисуе (Seth Tisue), Андрей Формига (Andrei Formiga), Дмитрий Григорьев (Dmitry Grigoriev), Джордж Бергер (George Berger), Говард Ловетт (Howard Lovatt), Джон П. Эйрих (John P. Eurich), Мариус Скуртеску (Marius Scurtescu), Джефф Эрвин (Jeff Ervin), Джейми Уэбб (Jamie Webb), Курт Зольман (Kurt Zoglmann), Дин Уэмплер (Dean Wampler), Николай Линдберг (Nikolaj Lindberg), Питер Маклейн (Peter McLain), Аркадиуш Стрыйски (Arkadiusz Stryjski), Шанки Сурана (Shanky Surana), Крейг Бордолон (Craig Bordelon), Александр Пэтри (Alexandre Patry), Филипп Моэнс (Filip Moens), Фред Янон (Fred Janon), Джефф Хеон (Jeff Heon), Борис Лорбер (Boris Lorbeer), Джим Менар (Jim Menard), Тим Аццопарди (Tim Azzopardi), Томас Юнг (Thomas Jung), Уолтер Чанг (Walter

Chang), Йерун Дийкмейер (Jeroen Dijkmeijer), Кейси Боумен (Casey Bowman), Мартин Смит (Martin Smith), Ричард Даллауэй (Richard Dallaway), Антони Стаббс (Antony Stubbs), Ларс Вестергрен (Lars Westergren), Маартен Хазевинкель (Maarten Hazewinkel), Мэтт Рассел (Matt Russell), Ремигиус Михаловский (Remigiusz Michalowski), Андрей Толопко (Andrew Tolopko), Кертис Стэнфорд (Curtis Stanford), Джошуа Каш (Joshua Cough), Земен Денг (Zemian Deng), Кристофер Родригес Масиас (Christopher Rodrigues Macias), Хуан Мигель Гарсия Лопес (Juan Miguel Garcia Lopez), Мишель Шинц (Michel Schinz), Питер Мур (Peter Moore), Рэндолф Кале (Randolph Kahle), Владимир Кельман (Vladimir Kelman), Даниэль Гронау (Daniel Gronau), Дирк Детеринг (Dirk Detering), Хироаки Накамура (Hiroaki Nakamura), Оле Хогаард (Ole Hougaard), Бхаскар Маддала (Bhaskar Maddala), Дэвид Бернар (David Bernard), Дерек Махар (Derek Mahar), Джордж Коллиас (George Kollias), Кристиан Нордал (Kristian Nordal), Нормен Мюллер (Normen Mueller), Рафаэль Феррейра (Rafael Ferreira), Бинил Томас (Binil Thomas), Джон Нильсон (John Nilsson), Хорхе Ортис (Jorge Ortiz), Маркус Шульте (Marcus Schulte), Вадим Герасимов (Vadim Gerasimov), Камерон Таггарт (Cameron Taggart), Джон-Андерс Тейген (Jon-Anders Teigen), Сильвестр Забала (Silvestre Zabala), Уилл Маккуин (Will McQueen) и Сэм Оуэн (Sam Owen).

Хочется также сказать спасибо тем, кто отправил сообщения о замеченных неточностях после публикации двух первых изданий. Это Феликс Зигрист (Felix Siegrist), Лотар Мейер-Лербс (Lothar Meyer-Lerbs), Диетард Михаэлис (Diethard Michaelis), Рошан Даврани (Roshan Dawrani), Донн Стефан (Donn Stephan), Уильям Утер (William Uther), Франсиско Ревербель (Francisco Reverbel), Джим Балтер (Jim Balter), Фрик де Брюйн (Freek de Bruijn), Амброз Лэнг (Ambrose Laing), Сехар Прабхала (Sekhar Prabhala), Левон Салдамли (Levon Saldamli), Эндрю Бурсавич (Andrew Bursavich), Хьялмар Петерс (Hjalmar Peters), Томас Фер (Thomas Fehr), Ален О'Ди (Alain O'Dea), Роб Диккенс (Rob Dickens), Тим Тейлор (Tim Taylor), Кристиан Штернагель (Christian Sternagel), Мишель

Паризьен (Michel Parisien), Джоэл Нили (Joel Neely), Брайан Маккеон (Brian McKeon), Томас Фер (Thomas Fehr), Джозеф Эллиотт (Joseph Elliott), Габриэль да Силва Рибейро (Gabriel da Silva Ribeiro), Пабло Рипольес (Pablo Ripolles), Дуглас Гейлор (Douglas Gaylor), Кевин Сквайр (Kevin Squire), Гарри-Антон Талвик (Harry-Anton Talvik), Кристофер Симпкинс (Christopher Simpkins), Мартин Витман-Функ (Martin Witmann-Funk), Джим Балтер (Jim Balter), Питер Фостер (Peter Foster), Крейг Бордолон (Craig Bordelon), Хайнц-Питер Гум (Heinz-Peter Gumm), Питер Чапин (Peter Chapin), Кевин Райт (Kevin Wright), Анантан Сринивасан (Ananthan Srinivasan), Омар Килани (Omar Kilani), Дон Стефан (Donn Stephan), Гюнтер Ваффлер (Guenther Waffler).

Лекс хотел бы поблагодарить специалистов, среди которых Аарон Абрамс (Aaron Abrams), Джейсон Адамс (Jason Adams), Генри и Эмили Крутчер (Henry and Emily Crutcher), Джои Гибсон (Joey Gibson), Гунар Хиллерт (Gunnar Hillert), Мэтью Линк (Matthew Link), Тоби Рейлтс (Toby Reyelts), Джейсон Снейп (Jason Snape), Джон и Мелинда Уэзерс (John and Melinda Weathers), и всех представителей Atlanta Scala Enthusiasts за множество полезных обсуждений конструкции языка, его математических основ и способов представления языка Scala специалистам-практикам.

Особую благодарность хочется выразить Дэйву Брикчетти (Dave Briccetti) и Адриану Мурсу (Adriaan Moors) за рецензирование третьего издания, а также Маркони Ланна (Marconi Lanna) не только за рецензирование, но и за его предложение написать книгу, поступившее после обсуждения обновлений, появившихся с момента выхода предыдущего издания.

Билл хотел бы поблагодарить нескольких специалистов за предоставление информации и советы по изданию книги. Его благодарность адресуется Гэри Корнеллу (Gary Cornell), Грегу Доенчу (Greg Doench), Энди Ханту (Andy Hunt), Майку Леонарду (Mike Leonard), Тайлеру Ортману (Tyler Ortman), Биллу Поллоку (Bill Pollock), Дейву Томасу (Dave Thomas) и Адаму Райту (Adam Wright). Билл также хотел бы поблагодарить Дика Уолла (Dick Wall) за

сотрудничество при разработке курса Escalate’s Stairway to Scala course, который большей частью основывался на материалах, вошедших в эту книгу. Наш многолетний опыт преподавания курса Stairway to Scala помог повысить качество книги. И наконец, Билл хотел бы выразить благодарность Дарлин Грюндль (Darlene Gruendl) и Саманте Вулф (Samantha Woolf) за помощь в завершении третьего издания.

Введение

Книга является справочным руководством по языку программирования Scala, созданному людьми, непосредственно занимающимися разработкой Scala. Мы преследовали цель научить вас всему необходимому для превращения в продуктивного программиста на языке Scala. Все примеры в издании скомпилированы с использованием Scala версии 2.11.7, за исключением тех, что имеют пометку 2.12, — они скомпилированы с использованием версии 2.12.0-M3.

Для кого предназначена эта книга

Книга в основном рассчитана на разработчиков, желающих научиться программировать или же создать очередной проект на Scala. Кроме того, издание должно заинтересовать тех, кто хочет расширить свой кругозор, изучив новые концепции. Если вы, к примеру, программируете на Java, то в этой книге найдете для себя множество понятий функционального программирования, а также передовых концепций из сферы объектно-ориентированного программирования. Мы уверены, что изучение Scala и заложенных в этот язык идей поможет вам повысить свой профессиональный уровень как программиста. Предполагается, что вы уже владеете общими знаниями в области программирования. Хотя язык Scala вполне подходит на роль первого изучаемого языка программирования, это не та книга, которая может использоваться для обучения программированию. В то же время от вас не требуется быть каким-то особенным знатоком языков программирования. Хотя большинство людей использует Scala на платформе Java, эта книга не предполагает, что вы хорошо знаете Java. Но все же мы ожидаем, что многим читателям Java известен, и поэтому иногда сравниваем Scala с Java, чтобы помочь таким читателям понять разницу.

Как пользоваться книгой

Рекомендуется читать книгу в порядке следования глав, от начала до конца. Мы очень старались в каждой главе вводить читателя в курс только одной темы и объяснять новые лишь в понятиях из ранее рассмотренных тем. Поэтому, если перескочить вперед, чтобы поскорее в чем-то разобраться, можно встретить темы, в которых используются еще не осмысленные вами понятия. Мы считаем, что приобретать знания в области программирования на Scala лучше постепенно, читая главы по порядку.

При встрече незнакомого понятия можно обратиться к предыдущему материалу. Многие читатели бегло просматривают части книги, и это вполне нормально. Но при встрече с незнакомыми терминами можно понять, что просмотр был слишком поверхностным, и вернуться к более раннему материалу.

Книга может послужить также в качестве справочника по языку. Конечно, существует официальная спецификация языка Scala, но в ней прослеживается стремление к точности в ущерб удобству чтения. Хотя в данной книге и не охвачены абсолютно все подробности языка Scala, его особенности изложены в ней вполне обстоятельно, и по мере освоения программирования на Scala она может стать доступным справочником по языку.

Как изучать Scala

Весьма обширные познания о языке Scala можно получить, просто прочитав книгу от начала до конца. Но можно освоить Scala быстрее и основательнее, если выполнить некоторые дополнительные действия.

Прежде всего предлагаем воспользоваться множеством примеров программ, включенных в эту книгу. В процессе их написания придется вдумываться в каждую строку кода. Попытки разнообразить код позволят вам сильнее заинтересоваться изучаемой темой и убедиться в том, что вы действительно поняли,

как этот код работает.

Не стоит обходить стороной и онлайн-форумы. Общение там позволит вам и многим другим приверженцам языка Scala помочь друг другу в его освоении. Существует множество рассылок, дискуссионных форумов, чатов, вики-источников и несколько посвященных Scala информационных каналов с соответствующими публикациями. Не пожалейте времени на подбор источников информации, которые в наибольшей степени отвечают вашим запросам. Вы станете меньше увязать в решении малозначительных проблем, высвободив тем самым больше времени для рассмотрения более серьезных и глубоких вопросов.

И наконец, получив при чтении книги достаточный объем знаний, приступайте к разработке собственного программного проекта. Поработайте с нуля над созданием какой-нибудь небольшой программы или напишите дополнение к более объемному приложению. Одним чтением быстрых результатов вы не добьетесь.

Условные обозначения

При первом упоминании какого-либо *понятия* или *термина* его название дается курсивом. Для небольших встроенных в текст примеров кода, таких как `x + 1`, используется моноширинный шрифт. Большие примеры кода представлены в виде отдельных блоков, которые также выделены моноширинным шрифтом:

```
def hello() = {
  println("Hello, world!")
}
```

Когда описывается работа с интерактивной оболочкой, ответы оболочки выделяются шрифтом на сером фоне:

```
scala> 3 + 4
```

```
res0: Int = 7
```

Код, который должен быть заменен подходящим по контексту значением, выделяется курсивом.

```
выбор match { варианты }
```

Помимо примеров вы также встретите подсказки, советы и справочную информацию, оформленные следующим образом.

примечание

Советы, подсказки, разъяснения оформляются вот так. Подобные блоки помогут вам лучше понимать концепции. Иногда в них мы также будем призывать вас обратить внимание на определенные ситуации, которых следует избегать.

Дополнения, которые выходят за рамки краткого совета или врезки, будут выглядеть немного иначе.

Справочная информация

Мы используем такое оформление, чтобы объяснить концепции, выходящие за рамки простой врезки. Здесь вы встретите подробности, которые не нужны для понимания материала книги, — они лишь дополняют определенную тему.

Обзор содержимого

- Глава 1 «Масштабируемый язык» представляет обзор конструкции языка Scala, а также связанные с ним логические обоснования и исторические события.

- Глава 2 «Первые шаги в Scala» показывает, как в языке решаются основные задачи программирования, не вдаваясь в подробности, касающиеся особенностей работы механизмов языка. Цель этой главы — начало практической работы по набору и запуску кода Scala.

- Глава 3 «Последующие шаги в Scala» демонстрирует более сложные функциональные возможности языка Scala. Изучив материал, вы сможете приступить к использованию Scala для написания сценариев, решающих простые задачи.

- Глава 4 «Классы и объекты» — это начало углубленного рассмотрения языка Scala с описанием его основных объектно-ориентированных строительных блоков и указаниями по выполнению компиляции и запуску приложений Scala.

- Глава 5 «Основные типы и операции» охватывает основные типы Scala, их литералы, операции, которые могут над ними проводиться, вопросы работы уровней приоритетности и ассоциативности и дает представление об обогащающих оболочках.

- Глава 6 «Функциональные объекты» углубляет представление об объектно-ориентированных свойствах Scala, используя в качестве примера функциональные (то есть неизменяемые) рациональные числа.

- Глава 7 «Встроенные структуры управления» показывает способы использования встроенных структур управления Scala `if`, `while`, `for`, `try` и `match`.

- Глава 8 «Функции и замыкания» углубленно рассматривает функции как основные строительные блоки функциональных языков.

- Глава 9 «Управляющие абстракции» демонстрирует пути

совершенствования основных управляющих структур Scala путем определения ваших собственных управляющих абстракций.

- Глава 10 «Композиция и наследование» рассматривает имеющуюся в Scala дополнительную поддержку объектно-ориентированного программирования. Затрагиваемые темы не носят такого же фундаментального характера, как те, что излагались в главе 4, но с рассматриваемыми в них вопросами часто приходится сталкиваться на практике.

- Глава 11 «Иерархия в Scala» объясняет иерархию наследования в языке Scala и рассматривает универсальные методы и низшие типы.

- Глава 12 «Трейты» охватывает существующий в Scala механизм создания подмешиваемых композиций. Показана работа трейтов, описываются примеры их наиболее частого использования и объясняется, как с помощью трейтов совершенствуется традиционное множественное наследование.

- Глава 13 «Пакеты и импортируемый код» рассматривает вопросы программирования в целом, включая высокоуровневые пакеты, инструкции импортирования и модификаторы управления доступом, подобные `protected` и `private`.

- Глава 14 «Утверждения и тесты» показывает существующий в Scala механизм утверждений и предоставляет обзор ряда инструментальных средств, доступных для написания тестов в Scala. Основное внимание уделено средству ScalaTest.

- Глава 15 «Case-классы и поиск по шаблону» вводит двойные конструкции, поддерживающие ваши действия при написании обычных, неинкапсулированных структур данных. Case-классы и поиск по шаблону принесут особую пользу в ходе работы с

древовидными рекурсивными данными.

- Глава 16 «Работа со списками» подробно рассматривает списки, которые можно отнести, наверное, к самым востребованным структурам данных в программах на Scala.

- Глава 17 «Работа с другими коллекциями» показывает способы использования основных коллекций Scala, таких как списки, массивы, кортежи, наборы и отображения.

- Глава 18 «Изменяемые объекты» объясняет суть изменяемых объектов и обеспечиваемого Scala синтаксиса для выражения этих объектов. Глава завершается практическим примером моделирования дискретного события, в котором показан ряд изменяемых объектов в действии.

- Глава 19 «Параметризация типов» на конкретных примерах объясняет некоторые из технологических приемов для сокрытия информации, представленных в главе 13, например создание класса для функциональных запросов. Материалы главы подводят к описанию вариантности параметров типа и способов их взаимодействия с сокрытием информации.

- Глава 20 «Абстрактные элементы» описывает все разновидности поддерживаемых Scala абстрактных элементов — не только методов, но также полей и типов, которые могут объявляться абстрактными.

- Глава 21 «Подразумеваемые преобразования и параметры» рассматривает две конструкции, способные помочь избавить исходный код от излишней детализации, позволяя предоставить ее самому компилятору.

- Глава 22 «Реализация списков» описывает реализацию класса `List`. В ней рассматривается работа списков в Scala, в которой важно разбираться. Кроме того, реализация списков показывает,

как используется ряд характерных особенностей языка Scala.

- Глава 23 «Возвращение к выражениям for» показывает, как выражения `for` превращаются в вызовы методов `map`, `flatMap`, `filter` и `foreach`.

- Глава 24 «Углубленное изучение коллекций» предлагает углубленный обзор библиотеки коллекций.

- Глава 25 «Архитектура коллекций Scala» демонстрирует, как устроена библиотека коллекций и как можно реализовывать собственные коллекции.

- Глава 26 «Экстракторы» показывает, как можно применять поиск по шаблонам в отношении не только case-классов, но и любых произвольных классов.

- Глава 27 «Аннотации» показывает, как можно работать с расширениями языка посредством аннотаций. В главе дается описание нескольких стандартных аннотаций и показано, как создать собственные аннотации.

- Глава 28 «Работа с XML» объясняет порядок обработки в Scala данных в формате XML. Показаны идиомы для создания XML, проведения синтаксического анализа данных в этом формате и обработки таких данных после анализа.

- Глава 29 «Модульное программирование с использованием объектов» содержит описание способов использования объектов Scala в качестве модульной системы.

- Глава 30 «Равенство объектов» высвечивает ряд проблемных вопросов, требующих рассмотрения при написании метода `equals`. Также описываются узкие места, которые следует обходить стороной.

- Глава 31 «Сочетание кодов Scala и Java» рассматривает проблемы, возникающие при сочетании совместно используемых кодов на Scala и Java в одном и том же проекте, и предлагает способы, позволяющие находить их решения.

- Глава 32 «Фьючерсы и многопоточные вычисления» показывает способы работы с классом Scala под названием `Future`. Хотя для программ на Scala могут использоваться примитивы многопоточных вычислений и библиотеки, применяемые на Java-платформе, фьючерсы помогут избежать возникновения условий взаимных блокировок и состязательных условий, являющихся проклятием традиционных подходов к многопоточным вычислениям в виде потоков и блокировок.

- Глава 33 «Синтаксический анализ с применением комбинаторов» показывает способы создания парсеров с использованием имеющейся в Scala библиотеки парсер-комбинаторов.

- Глава 34 «Программирование GUI» предлагает краткий обзор библиотеки Scala, упрощающей GUI-программирование с применением среды Swing.

- Глава 35 «Электронная таблица SCells» связывает все вместе, показывая полнофункциональное приложение электронной таблицы, написанное на языке Scala.

Ресурсы

Самые последние выпуски Scala, ссылки на документацию и ресурсы сообщества можно найти на сайте Scala по адресу: **http://www.scala-lang.org**.

Исходный код

Исходный код, рассматриваемый в данной книге, выпущенный под открытой лицензией в виде ZIP-файла, можно найти на сайте книги: **http://booksites.artima.com/programming_in_scala_3ed**.

1. Масштабируемый язык

Название Scala расшифровывается как «масштабируемый язык». Язык получил такое название, поскольку рассчитан на наращивание нагрузок по мере увеличения пользовательских потребностей. Язык Scala можно использовать для решения широкого круга задач программирования: от написания небольших сценариев до создания больших систем[1].

Освоить Scala нетрудно. Он запускается на стандартной Java-платформе и полноценно взаимодействует со всеми Java-библиотеками. Этот язык хорошо подходит для написания сценариев, объединяющих Java-компоненты. Но еще лучше его эффективность может раскрыться при создании больших систем и сред многократно используемых компонентов.

С технической точки зрения Scala является смесью объектно-ориентированной и функциональной концепций программирования в статически типизированном языке. Сплав объектно-ориентированного и функционального программирования проявляется во многих аспектах Scala — он, вероятно, может считаться более всеобъемлющим, чем другие популярные языки. Когда дело доходит до масштабируемости, два стиля программирования приобретают взаимодополняющую силу. Используемые в Scala конструкции функционального программирования упрощают ускоренное создание интересных компонентов из простых частей. Его объектно-ориентированные конструкции облегчают структурирование больших систем и их адаптацию к новым требованиям. Сочетание двух стилей в Scala позволяет создавать новые виды шаблонов программирования и абстракций компонентов. Оно также способствует выработке понятных и лаконичных стилей программирования. И благодаря такой податливости языка программирование на Scala может принести массу удовольствия.

Эта вступительная глава отвечает на вопрос: «А почему именно

Scala?». В ней дается общий обзор конструкции Scala и ее обоснования. Прочитав главу, вы получите базовое представление о том, что такое Scala и с какого рода задачами этот язык поможет вам справиться. Хотя сама книга представляет собой руководство по языку Scala, данную главу нельзя считать частью этого руководства. И если вам не терпится приступить к написанию кода на языке Scala, можете сразу перейти к изучению главы 2.

1.1. Язык, наращиваемый под ваши потребности

Программы различных размеров требуют, как правило, использования различных программных конструкций. Рассмотрим, к примеру, следующую небольшую программу на Scala:

```
var capital = Map("US" -> "Washington", "France" -
> "Paris")
capital += ("Japan" -> "Tokyo")
println(capital("France"))
```

Эта программа устанавливает отображение стран на их столицы, модифицирует его, добавляя новую конструкцию (`"Japan" -> "Tokyo"`), и выводит название столицы, связанное со страной `France`[2]. В этом примере используется настолько высокоуровневая система записи, что она не загромождена ненужными запятыми и сигнатурами типов. И действительно, возникает ощущение использования современного сценарного языка вроде Perl, Python или Ruby. Одной из общих их характеристик, применимых к данному примеру, является поддержка всеми ими в синтаксисе языка конструкции ассоциативного отображения.

Ассоциативные отображения очень полезны, поскольку помогают поддерживать разборчивость и краткость программ. Но порой можно не согласиться с их философией «одного

подходящего абсолютно всем размера», так как вам в своей программе необходимо более тонко управлять свойствами отображений. Scala, если требуется, обеспечивает вам точное управление, поскольку отображения в нем не являются синтаксисом языка. Это библиотечные абстракции, которые можно расширять и приспосабливать под свои нужды.

В показанной ранее программе вы получите исходную реализацию отображения `Map`, но ее можно будет без особого труда изменить. К примеру, допустимо указать конкретную реализацию, такую как `HashMap` или `TreeMap`, или вызвать метод `par` для получения отображения `ParMap`, операции в котором выполняются в параллельном режиме. Можно указать для отображения значение по умолчанию или переопределить любой другой метод созданного вами отображения. Во всех случаях для отображений разрешено использовать такой же простой синтаксис доступа, как и в приведенном примере.

В нем показано, что Scala может обеспечить вам как удобство, так и гибкость. В Scala имеется набор удобных конструкций, помогающих быстро запустить проект и позволяющих программировать в приятном лаконичном стиле. В то же время есть гарантии, что вы не сможете «перерасти» язык. Вы всегда сможете перекроить программу под свои требования, поскольку все в ней основано на библиотечных модулях, которые можно выбрать и приспособить под свои нужды.

Выведение новых типов

Эрик Рэймонд (Eric Raymond) ввел в качестве двух метафор разработки программных продуктов собор и базар[3]. Под собором понимается почти идеальная разработка, для создания которой требуется много времени. После сборки она долго остается неизменной. В отличие от этого, базар разработчики подстраивают и дополняют каждый день. В работе Рэймонда базар является метафорой, описывающей разработку программного обеспечения с

открытым кодом. Гай Стил (Guy Steele) отметил в докладе о наращивании языка, что аналогичное различие может быть применено к разработке языка программирования[4]. Scala больше похож на базар, чем на собор, в том смысле, что разработан с расчетом на то, что его смогут расширять и адаптировать те, кто на нем программирует. Вместо предоставления всех конструкций, которые только могут пригодиться в одном всеобъемлющем языке, Scala вкладывает инструменты для создания таких конструкций в ваши руки.

Рассмотрим пример. Многие приложения нуждаются в целочисленном типе, который при выполнении арифметических операций может становиться произвольно большим без переполнения или циклического перехода в начало. В Scala такой тип задается в библиотеке класса `scala.BigInt`. Определение использующего этот тип метода, который вычисляет факториал переданного ему целочисленного значения, имеет следующий вид[5]:

```
def factorial(x: BigInt): BigInt =
    if (x == 0) 1 else x * factorial(x - 1)
```

Теперь, вызвав `factorial(30)`, вы получите:

```
265252859812191058636308480000000
```

`BigInt` похож на встроенный тип, поскольку со значениями этого типа можно использовать целочисленные литералы и операторы вроде * и –. Тем не менее это просто класс, определение которого задано в стандартной библиотеке Scala[6]. Если бы класса не было, любой программист на Scala мог бы запросто написать его реализацию, например путем создания оболочки для имеющегося в языке Java класса `classjava.math.BigInteger` (фактически именно так и реализован класс `BigInt` в Scala).

Конечно же, класс Java можно использовать напрямую. Но результат будет не столь приятным: хоть Java и позволяет вам

создавать новые типы, они не производят впечатления получающих естественную поддержку языка:

```
import java.math.BigInteger

def factorial(x: BigInteger): BigInteger =
    if (x == BigInteger.ZERO)
        BigInteger.ONE
    else
        x.multiply(factorial(x.subtract(BigInteger.
```

`BigInt` является представителем многих других числовых типов — больших десятичных, комплексных чисел, рациональных чисел, доверительных интервалов, полиномов, и этот список можно продолжить. В некоторых языках программирования часть этих типов реализуется естественным образом. Например, в Lisp, Haskell и Python реализуются большие целые числа, в Fortran и Python — комплексные числа. Но любой язык, в котором предпринимаются попытки одновременно задействовать все эти абстракции, разрастается до таких размеров, что становится неуправляемым. Более того, даже если бы такой язык существовал, некоторые приложения, несомненно, выиграли бы из-за других числовых типов, связанных с числами, которые им не поставлялись бы. Следовательно, подход, при котором предпринимается попытка реализовать все в одном языке, не позволяет получить хорошую масштабируемость. С помощью Scala, напротив, пользователи могут наращивать и адаптировать его в нужных направлениях, определяя несложные в использовании библиотеки, которые производят впечатление средств, получающих естественную поддержку языка.

Выведение новых управляющих конструкций

Предыдущий пример показывает, что Scala позволяет добавлять

новые типы, использовать которые так же удобно, как и встроенные. Тот же принцип расширения применяется и к управляющим структурам. Этот вид расширения иллюстрируется с помощью Akka — API Scala для основанного на использовании акторов многопоточного программирования.

Поскольку распространение многоядерных процессоров в ближайшие годы продолжится, для достижения приемлемой производительности все чаще может требоваться использование более высокой степени распараллеливания ваших приложений. Зачастую это может повлечь за собой переписывание кода, чтобы распределить вычисление по нескольким параллельным потокам. К сожалению, на практике создание надежных многопоточных приложений является весьма непростой задачей. Модель потоковой обработки в Java выстроена вокруг совместно используемой памяти и блокировок и зачастую трудно поддается осмыслению, особенно по мере масштабирования систем с возрастанием их объема и сложности. Обеспечить отсутствие состязательных условий или скрытых взаимных блокировок, не выявляемых в ходе тестирования, но способных проявиться в ходе эксплуатации, довольно трудно. Возможно, более безопасным альтернативным решением станет использование архитектуры передачи сообщений, подобной применению акторов в языке программирования Erlang.

Java поставляется с богатой библиотекой параллельных вычислений, основанной на применении потоков. Программы на Scala могут использовать ее точно так же, как и любой другой API Java. Но есть еще Akka — дополнительная библиотека Scala, реализующая модель акторов, похожую на ту, что используется в Erlang.

Акторы являются абстракциями параллельных вычислений, которые могут быть реализованы в качестве надстроек над потоками. Они обмениваются данными, отправляя друг другу сообщения. Актор может выполнить две основные операции: отправку сообщения и его получение. Операция отправки,

обозначаемая восклицательным знаком (`!`), отправляет сообщение актору. Пример, в котором фигурирует актор по имени `recipient`, выглядит следующим образом:

```
recipient ! msg
```

Отправка осуществляется в асинхронном режиме, то есть отправляющий сообщение актор может продолжить выполнение кода, не дожидаясь получения и обработки сообщения. У каждого актора имеется *почтовый ящик*, в котором входящие сообщения выстраиваются в очередь. Актор обрабатывает сообщения, поступающие в его почтовый ящик, в блоке получения `receive`:

```
def receive = {
    case Msg1 => ... // обработка Msg1
    case Msg2 => ... // обработка Msg2
    // ...
}
```

Блок `receive` состоит из нескольких инструкций выбора `case`, в каждой из которых содержится запрос к почтовому ящику на соответствие шаблону сообщения. В почтовом ящике выбирается первое же сообщение, соответствующее условию какой-либо инструкции выбора, и в отношении него выполняется какое-либо действие. Когда в почтовом ящике не оказывается сообщений, актор приостанавливает работу и ожидает дальнейшего поступления сообщений.

В качестве примера рассмотрите простой Akka-актор, реализующий сервис подсчета контрольной суммы:

```
class ChecksumActor extends Actor {
    var sum = 0
    def receive = {
        case Data(byte) => sum += byte
        case GetChecksum(requester) =>
```

```
                val checksum = ~(sum & 0xFF) + 1
                requester ! checksum
        }
    }
```

Сначала в акторе определяется локальная переменная `sum` с начальным нулевым значением. Затем — блок получения `receive`, который будет обрабатывать сообщения. При получении `Data`-сообщения он прибавляет значение содержащегося в нем аргумента `byte` к значению переменной `sum`. При получении сообщения `GetChecksum` он вычисляет контрольную сумму из текущего значения переменной `sum` и отправляет результат обратно в адрес пославшего запрос `requester`, используя код отправки сообщения `requester ! checksum`. Поле `requester` встроено в сообщение `GetChecksum`, обычно оно ссылается на актор, пославший запрос.

Пока что мы не ждем от вас полного понимания примера использования актора. Достаточно того, что существенным обстоятельством для темы масштабируемости в этом примере служит тот факт, что ни блок `receive`, ни инструкция отправки сообщения (`!`) не являются встроенными операциями Scala. Даже при том что внешний вид и работа блока `receive` во многом напоминают встроенную управляющую конструкцию, на самом деле это метод, определенный в библиотеке акторов Akka. Аналогично этому, даже при том что конструкция `!` похожа на встроенный оператор, она также является всего лишь методом, определенным в библиотеке акторов Akka. Обе эти конструкции абсолютно независимы от языка программирования Scala.

Синтаксис блока `receive` и оператора отправки (`!`) выглядит в Scala во многом так же, как в Erlang, но в Erlang эти конструкции встроены в язык. В Akka также реализуется большинство других конструкций многопоточного программирования, имеющихся в Erlang, например конструкции отслеживания сбойных акторов и

истечения времени ожидания. В целом модель акторов показала себя весьма удачным средством для выражения многопоточных и распределенных вычислений. Несмотря на то что акторы должны быть определены в библиотеке, они могут рассматриваться как составная часть языка Scala.

Этот пример иллюстрирует возможность наращивания языка Scala в новых направлениях, даже в таких специализированных, как программирование многопоточных приложений. Разумеется, для этого понадобятся квалифицированные проектировщики и программисты. Но важен сам факт наличия такой возможности: вы можете проектировать и реализовывать в Scala абстракции, которые относятся к принципиально новым прикладным областям, но воспринимаются при использовании как естественно поддерживаемые самим языком.

1.2. Что делает Scala масштабируемым языком

На возможность масштабирования влияет множество факторов: от особенностей синтаксиса до абстрактных конструкций компонентов. Но если бы потребовалось назвать всего один аспект Scala, способствующий масштабируемости, мы бы выбрали присущее этому языку сочетание объектно-ориентированного и функционального программирования (мы немного слукавили, на самом деле это два аспекта, но они взаимосвязаны).

Scala пошел дальше всех остальных широко известных языков в объединении объектно-ориентированного и функционального программирования в однородную конструкцию языка. Например, там, где в других языках объекты и функции — это два разных понятия, в Scala функция по смыслу является объектом. Типы функций представляют собой классы, которые могут наследоваться подклассами. Это может показаться не более чем теоретическими особенностями, но они имеют серьезные последствия для возможностей масштабирования. Фактически ранее упомянутое понятие актора не может быть реализовано без этой унификации

функций и объектов. В данном разделе дается обзор примененных в Scala способов смешивания объектно-ориентированной и функциональной концепций.

Scala — объектно-ориентированный язык

Объектно-ориентированное программирование развивалось весьма успешно. Появившись в языке Simula в середине 1960-х годов и Smalltalk в 1970-х годах, оно теперь доступно в подавляющем большинстве языков. В некоторых областях все полностью захвачено объектами. Хотя точного определения того, что такое объектно-ориентированные средства, не существует, это явно касается объектов, интересующих программистов.

В принципе, мотивация для применения объектно-ориентированного программирования очень проста: всё, за исключением самых простых программ, нуждается в определенной структуре. Наиболее понятный путь достижения желаемого результата заключается в помещении данных и операций в своеобразные контейнеры. Основной замысел объектно-ориентированного программирования состоит в придании этим контейнерам полной универсальности, чтобы в них могли содержаться не только операции, но и данные, и чтобы сами они также были элементами, которые могли бы храниться в других контейнерах или передаваться операциям в качестве параметров. Такие контейнеры называются объектами. Алан Кей (Alan Kay), создатель языка Smalltalk, заметил, что таким образом простейший объект имеет такой же принцип построения, что и полноценный компьютер: под формализованным интерфейсом данные в нем сочетаются с операциями[7]. То есть объекты имеют непосредственное отношение к масштабируемости языка: одни и те же технологии применяются к построению как малых, так и больших программ.

Хотя долгое время объектно-ориентированное программирование являлось преобладающей тенденцией,

немногие языки стали последователями Smalltalk во внедрении этого принципа построения в свое логическое решение. Например, многие языки допускают использование элементов, не являющихся объектами, — можно вспомнить имеющиеся в языке Java значения элементарных типов. Или же в них допускается применение статических полей и методов, не входящих в какой-либо объект. Эти отклонения от чистой идеи объектно-ориентированного программирования на первый взгляд выглядят вполне безобидными, но у них имеется досадная тенденция к усложнению и ограничению масштабирования.

В отличие от этого, язык Scala является объектно-ориентированным в чистом виде: каждое значение есть объект, и каждая операция — вызов метода. Например, когда в Scala заходит речь о вычислении `1 + 2`, фактически вызывается метод по имени `+`, который определен в классе `Int`. Можно определять методы с именами, похожими на операторы, а клиенты вашего API смогут воспользоваться этими методами при записи оператора. Именно так разработчик применяемого в Akka API акторов позволил вам воспользоваться выражением `requester ! sum`, показанным в предыдущем примере: `!` является методом класса `Actor`.

Когда речь заходит о составлении объектов, Scala проявляется как более совершенный язык по сравнению с большинством других. Примером могут служить имеющиеся в Scala *трейты*. Они подобны интерфейсам в Java, но могут содержать также реализации методов и даже поля[8]. Объекты создаются путем *смешивания состава*, при котором к элементам класса добавляются элементы ряда трейтов. Таким образом, различные аспекты классов могут быть инкапсулированы в различных трейтах. Это выглядит как множественное наследование, но отличается в конкретных деталях. В отличие от класса, трейт может добавить в родительский класс новые функциональные возможности, что придает трейтам более высокую степень подключаемости по сравнению с классами. В частности, благодаря этому удается избежать возникновения присущих множественному

наследованию классических проблем «алмазного» наследования, которые возникают, когда один и тот же класс наследуется по нескольким различным путям.

Scala — функциональный язык

Хотя Scala является чистым объектно-ориентированным языком, его можно назвать и полноценным функциональным языком. Идеи функционального программирования старше электронных вычислительных систем. Их основы были заложены в лямбда-исчислениях Алонзо Черча (Alonzo Church), разработанных в 1930-е годы. Первым языком функционального программирования был Lisp, появление которого датируется концом 1950-х годов. К другим популярным функциональным языкам относятся Scheme, SML, Erlang, Haskell, OCaml и F#. Долгое время функциональное программирование было на второстепенных ролях — будучи популярным в научных кругах, оно не столь широко использовалось в промышленности. Но в последние годы интерес к языкам и технологиям функционального программирования растет.

Функциональное программирование базируется на двух основных идеях. Первая заключается в том, что функции выступают как значения первого класса. В функциональных языках функция является значением, имеющим такой же статус, как целое число или строка. Функции можно передавать в качестве аргументов другим функциям, возвращать их как результаты из других функций или сохранять в переменных. Функцию можно также задавать внутри другой функции точно так же, как это делается при определении внутри функции целочисленного значения. И функции можно определять, не присваивая им имен, разбрасывая код с функциональными литералами с такой же легкостью, как и при написании целочисленного литерала, к примеру 42.

Функции как значения первого класса являются удобным

средством абстрагирования, касающегося операций и создания новых управляющих структур. Эта их универсальность обеспечивает более высокую степень выразительности, что зачастую приводит к созданию весьма разборчивых и кратких программ. Она также играет важную роль в обеспечении масштабируемости. В качестве примера библиотека тестирования ScalaTest предлагает конструкцию `eventually`, получающую функцию в качестве аргумента. Эта конструкция используется следующим образом:

```
val xs = 1 to 3
val it = xs.iterator
eventually { it.next() shouldBe 3 }
```

Код внутри `eventually`, являющийся утверждением `it.next() shouldBe 3`, включает в себя функцию, передаваемую невыполненной в метод `eventually`. Через настраиваемый период времени `eventually` начнет неоднократно выполнять функцию до тех пор, пока утверждение не будет успешно подтверждено.

В отличие от этого, в большинстве традиционных языков функции не являются значениями. Языки, дающие оценку функциям, относят их статус ко второму классу. Например, указатели на функции в C и C++ не имеют в этих языках такого же статуса, как другие, не относящиеся к функциям значения: указатели на функции могут только ссылаться на глобальные функции, не позволяя определять вложенные функции первого класса, имеющие некоторые значения в их среде. Они также не позволяют определять литералы безымянных функций.

Вторая основная идея функционального программирования заключается в том, что операции программы должны преобразовать входные значения в выходные, а не изменять данные по месту. Чтобы понять разницу, рассмотрим реализацию строк в Ruby и Java. В Ruby строка является массивом символов.

Символы в строке могут быть изменены по отдельности. Например, внутри одного и того же строкового объекта символ точки с запятой в строке можно заменить на точку. А в Java и Scala строка является последовательностью символов в математическом смысле. Замена символа в строке с использованием выражения вида `s.replace(';', '.')` приводит к возникновению нового строкового объекта, отличающегося от `s`. То же самое можно сказать по-другому: в Java строки неизменяемые, а в Ruby — изменяемые. То есть, рассматривая только строки, можно прийти к выводу, что Java является функциональным языком, а Ruby — нет. Неизменяемая структура данных является одним из краеугольных камней функционального программирования. В библиотеках Scala в качестве надстроек над соответствующими API Java определяется также множество других неизменяемых типов данных. Например, в Scala имеются неизменяемые списки, кортежи, отображения и наборы.

Еще один способ утверждения второй идеи функционального программирования заключается в том, что у методов не должно быть никаких побочных эффектов. Они должны обмениваться данными со своей средой только путем получения аргументов и возвращения результатов. Например, под это описание подпадает метод `replace`, принадлежащий Java-классу `String`. Он получает строку и два символа и выдает новую строку, где все появления одного символа заменены появлениями второго символа. Других эффектов от вызова `replace` нет. Методы, подобные `replace`, называются *референциально прозрачными*. Это означает, что для любого заданного ввода вызов функции может быть заменен его результатом, при этом семантика программы оказывается неизменной.

Функциональные языки заставляют применять неизменяемые структуры данных и референциально прозрачные методы. В некоторых функциональных языках это выражено в виде категоричных требований. Scala же дает возможность выбора. При желании можно писать программы в императивном стиле — так

называется программирование с изменяемыми данными и побочными эффектами. Но если нужно, то в большинстве случаев Scala позволяет с легкостью избежать применения императивных конструкций благодаря существованию хороших функциональных альтернатив.

1.3. Почему именно Scala

Подойдет ли вам язык Scala? Разбираться и принимать решение придется самостоятельно. По нашему мнению, помимо хорошей масштабируемости существует еще множество причин, по которым вам может понравиться программирование на Scala. В этом разделе будут рассмотрены четыре наиболее важных аспекта: совместимость, лаконичность, абстракции высокого уровня и расширенная статическая типизация.

Scala обладает совместимостью

Scala не требует резко отходить от платформы Java, чтобы продвинуться на шаг вперед от языка Java. Этот язык позволяет повышать ценность уже существующего кода, то есть опираться на то, что у вас уже есть, поскольку он был разработан для достижения беспрепятственной совместимости с Java9. Программы на Scala компилируются в байт-коды виртуальной машины Java (JVM). Производительность при выполнении этих кодов находится на одном уровне с производительностью программ на Java. Код Scala может вызывать методы Java, обращаться к полям Java, поддерживать наследование из классов Java и реализовывать интерфейсы Java. Для всего перечисленного не требуется ни специального синтаксиса, ни явных описаний интерфейса, ни какого-либо связующего кода. По сути, весь код Scala интенсивно использует библиотеки Java, зачастую даже без ведома программистов.

Еще одним аспектом полной совместимости является

интенсивное заимствование в Scala типов данных Java. Данные типа `Int` в Scala представлены в виде имеющегося в Java элементарного целочисленного типа `int`, соответственно, `Floats` представлены как `floats`, `Booleans` — как `booleans` и т. д. Массивы Scala отображаются на массивы Java. В Scala из Java позаимствованы также многие стандартные библиотечные типы. Например, тип строкового литерала `"abc"` в Scala фактически представлен классом `java.lang.String`, а выданное исключение должно быть подклассом `java.lang.Throwable`.

Java-типы в Scala не только заимствованы, но и «принаряжены» для придания им привлекательности. Например, строки в Scala поддерживают такие методы, как `toInt` или `toFloat`, которые преобразуют строки в целое число или число с плавающей точкой. То есть вместо `Integer.parseInt(str)` вы можете написать `str.toInt`. Как такого можно добиться без нарушения совместимости? Ведь в Java-классе `String` нет метода `toInt`! По сути, в Scala имеется универсальное решение для устранения противоречия между расширенным исполнением библиотеки и совместимостью. Scala позволяет определять *подразумеваемые преобразования*, которые всегда применяются при несовпадении типов или выборе несуществующих элементов. В рассматриваемом случае при поиске метода `toInt` для работы со строковым значением компилятор Scala не найдет такого элемента в классе `String`, но отыщет подразумеваемое преобразование, превращающее Java-класс `String` в экземпляр Scala-класса `StringOps`, в котором такой элемент определен. Затем преобразование автоматически применится, прежде чем будет выполнена операция `toInt`.

Код Scala также может быть вызван из кода Java. Иногда при этом следует учитывать некоторые нюансы. Scala — более богатый язык, чем Java, поэтому некоторые расширенные функции Scala должны быть закодированы, прежде чем они могут быть отображены на код Java. Подробности представлены в главе 31.

Scala лаконичен

Программы на Scala, как правило, отличаются краткостью. Программисты, работающие с этим языком, отмечают сокращение количества строк почти на порядок по сравнению с Java. Но это можно считать крайним случаем. Более консервативные оценки свидетельствуют о том, что обычная программа на Scala должна умещаться в половину тех строк, которые используются для аналогичной программы на Java. Меньшее количество строк означает не только сокращение объема набираемого текста, но и экономию сил при чтении и осмыслении программ, а также уменьшение числа возможных недочетов. Свой вклад в сокращение количества строк кода вносят сразу несколько факторов.

В синтаксисе Scala не используются некоторые шаблонные элементы, отягощающие программы на Java. Например, в Scala не обязательно применять точки с запятыми. Есть также несколько других областей, где синтаксис Scala менее зашумлен. В качестве примера можно сравнить, как записывается код классов и конструкторов в Java и Scala. В Java класс с конструктором зачастую выглядит так:

```
// это Java
class MyClass {
    private int index;
    private String name;
         public MyClass(int index, String name) {
        this.index = index;
        this.name = name;
    }
}
```

А в Scala, скорее всего, будет использоваться следующая запись:

```
class MyClass(index: Int, name: String)
```

Получив этот код, компилятор Scala создаст класс с двумя закрытыми переменными экземпляра (типа `Int` по имени `index` и типа `String` по имени `name`) и конструктор, который получает исходные значения для этих переменных в виде параметров. Код этого конструктора проинициализирует две переменные экземпляра значениями, переданными в качестве параметров. Короче говоря, в итоге вы получите те же функциональные свойства, что и у более многословной версии кода на Java10. Класс в Scala быстрее пишется и проще читается, а еще — и это наиболее важно — допустить ошибку при его создании значительно труднее, чем при создании класса в Java.

Еще одним фактором, способствующим лаконичности, является характерная для Scala подразумеваемость типа. Повторяющуюся информацию о типе можно отбросить, и тогда программы избавятся от лишнего и их легче будет читать.

Но, наверное, наиболее важным аспектом сокращения объема кода является наличие кода, не требующего внесения в программу, поскольку это уже сделано в библиотеке. Scala предоставляет вам множество инструментальных средств для определения эффективных библиотек, позволяющих выявить и вынести за скобки общее поведение. Например, различные свойства библиотечных классов могут быть выделены в трейты, которые способны перемешиваться друг с другом произвольным образом. Или же библиотечные методы могут быть параметризованы с использованием операций, позволяя вам определять конструкции, которые, по сути, являются вашими собственными управляющими структурами. Собранные вместе, эти конструкции дают возможность определять удобные в работе библиотеки.

Scala принадлежит к языкам высокого уровня

Программисты постоянно борются со сложностями. Для продуктивного программирования нужно понимать код, над которым вы работаете. Чрезмерно сложный код был причиной

краха многих программных проектов. К сожалению, важные - программные продукты обычно бывают многокомпонентными. Избежать этого невозможно, но ситуацией можно управлять.

Scala позволяет повышать уровень абстракции в разрабатываемых и используемых интерфейсах. Представим, к примеру, что имеется переменная `name` типа `String` и нужно определить, есть ли в этой строковой переменной символ в верхнем регистре. До выхода Java 8 приходилось создавать такой цикл:

```
boolean nameHasUpperCase = false; // Это Java
for (int i = 0; i < name.length(); ++i) {
    if (Character.isUpperCase(name.charAt(i))) {
        nameHasUpperCase = true;
        break;
    }
}
```

А в Scala можно написать следующий код:

```
val nameHasUpperCase = name.exists(_.isUpper)
```

Код Java считает строки низкоуровневыми элементами, требующими посимвольного перебора в цикле. Код Scala рассматривает те же самые строки как высокоуровневые последовательности символов, в отношении которых можно применять запросы с *предикатами*. Несомненно, код Scala намного короче и, обладая тренированным взглядом, его проще понять, чем код Java. Следовательно, вклад кода Scala в общую сложность приложения значительно меньше. Кроме того, уменьшается вероятность допустить ошибку.

Предикат `_.isUpper` является примером используемого в Scala функционального литерала[11]. В нем дается описание функции, получающей аргумент в виде символа (представленного знаком

подчеркивания) и проверяющей, не является ли этот символ буквой в верхнем регистре12.

В Java 8 появилась поддержка *лямбда-выражений* и *потоков* (streams), позволяющая выполнять подобные операции на Java. Вот как это могло бы выглядеть:

```
boolean nameHasUpperCase = // Это Java 8
    name.chars().anyMatch(
          (int ch) -> Character.isUpperCase((char)
ch)
);
```

Несмотря на существенное улучшение по сравнению с более ранними версиями Java, код Java 8 все же более многословен, чем его эквивалент на языке Scala. Излишняя тяжеловесность кода Java, а также давняя традиция использования в этом языке циклов могут многих Java-программистов натолкнуть на мысль о необходимости применения новых методов, подобных `exists`, позволяющих просто переписать циклы и смириться с растущей сложностью кода.

В то же время функциональные литералы в Scala действительно воспринимаются довольно легко и используются очень часто. По мере углубления знакомства со Scala перед вами будет открываться все больше и больше возможностей для определения и использования собственных управляющих абстракций. Вы поймете, что это поможет избежать повторений в коде, сохраняя лаконичность и чистоту программ.

Scala является статически типизированным языком

Системы со статической типизацией классифицируют переменные и выражения в соответствии с видом хранящихся и вычисляемых значений. Scala выделяется как язык своей весьма совершенной системой статической типизации. Обладая системой вложенных типов классов, во многом похожей на имеющуюся в Java, этот язык

позволяет вам проводить параметризацию типов с помощью средств *обобщенного программирования*, комбинировать типы с использованием *пересечений* и скрывать особенности типов, применяя *абстрактные типы*13. Таким образом формируется прочный фундамент для создания собственных типов, позволяющий разрабатывать безопасные и в то же время гибкие в использовании интерфейсы.

Если вам нравятся динамические языки, такие как Perl, Python, Ruby или Groovy, вы можете посчитать немного странным то, что система статических типов в Scala упоминается как одна из его сильных сторон. Ведь отсутствие системы статических типов часто называют основным преимуществом динамических языков. Наиболее часто, говоря о недостатках системы статических типов, приводят такие аргументы, как присущая программам многословность, воспрепятствование свободному самовыражению программистов и невозможность применения конкретных шаблонов динамических изменений программных систем. Но зачастую эти аргументы направлены не против идеи статических типов в целом, а против конкретных систем типов, воспринимаемых как слишком многословные или недостаточно гибкие. Например, Алан Кей, разработчик языка Smalltalk, однажды заметил: «Я не против типов, но мне не знакома ни одна система типов, которая не вызывала бы боли. Так что мне все еще нравится динамическая типизация»14.

В этой книге я надеюсь убедить вас в том, что система типов в Scala далека от вызывающей боль. На самом деле она вполне изящно справляется с двумя обычными опасениями, связываемыми со статической типизацией. Многословия удается избежать за счет логического вывода типа, а гибкость достигается благодаря поиску по шаблону и ряду новых способов записи и составления типов. С устранением этих препятствий к классическим преимуществам систем статических типов начинают относиться намного благосклоннее. Среди наиболее важных плюсов можно назвать проверяемые свойства программных

абстракций, безопасную реструктуризацию и более качественное документирование.

Проверяемые свойства. Системы статических типов способны подтверждать отсутствие конкретных ошибок, выявляемых в ходе выполнения программы. Это могут быть следующие свойства: булевы значения никогда не складываются с целыми числами; закрытые переменные недоступны за пределами своего класса; функции применяются к надлежащему количеству аргументов; к набору строк можно добавлять только строки.

Существующие в настоящее время системы статических типов ошибки других видов не выявляют. Например, обычно они не обнаруживают бесконечные функции, нарушение границ массивов или деление на нуль. Они также не смогут определить несоответствие вашей программы ее спецификации (при наличии таковой!). Поэтому некоторые отказываются от систем статических типов, считая их не слишком полезными. Аргументация следующая: если такие системы типов могут выявлять только простые ошибки, а блочные тесты обеспечивают более широкий охват, зачем вообще связываться со статическими типами? Мы считаем, что в этих аргументах упущено главное. Хотя система статических типов, конечно же, не может заменить собой блочное тестирование, она способна сократить количество необходимых блочных тестов, выявляя свойства, которые в противном случае нужно было бы протестировать. А блочное тестирование не заменит статическую типизацию. Ведь Эдсгер Дейкстра (Edsger Dijkstra) сказал, что тестирование позволяет убедиться лишь в наличии ошибок, но не в их отсутствии[15].

Безопасная реструктуризация. Системы статических типов дают гарантии, позволяющие вносить изменения в основной код, будучи совершенно уверенными в благополучном исходе этого действия. Рассмотрим, к примеру, реструктуризацию, при которой к методу добавляется еще один параметр. В статически типизированном языке можно вносить изменение, проводить перекомпиляцию системы и просто править те строки, которые

вызывают ошибку типа. Когда этот процесс завершится, вы будете уверены, что были найдены все места, требовавшие изменений. То же самое справедливо для другой простой реструктуризации, например изменения имени метода или перемещения метода из одного класса в другой. Во всех случаях проверка статического типа дает вполне достаточную уверенность в том, что работоспособность новой системы осталась на уровне работоспособности старой.

Документирование. Статические типы являются документацией программы, проверяемой компилятором на корректность. В отличие от обычного комментария, аннотация типа никогда не может быть устаревшей (по крайней мере, если содержащий ее исходный файл недавно успешно прошел компиляцию). Более того, компиляторы и интегрированные среды разработки (integrated development environments (IDE)) могут использовать аннотации для выдачи более качественной контекстной справки. Например, IDE может вывести на экран все доступные элементы путем определения статического типа выражения, которое выбирают, и дать возможность просмотреть все элементы этого типа.

Хотя статические типы в целом полезны для документирования программы, иногда они могут вызывать раздражение тем, что засоряют ее. Обычно удобным считается документирование тех сведений, которые читателям программы самостоятельно извлечь довольно трудно. Полезно знать, что в методе, который определен следующим образом:

```
def f(x: String) = ...
```

аргументы метода `f` должны принадлежать типу `String`. В то же время может вызвать раздражение по крайней мере одна из двух аннотаций в следующем примере:

```
val x: HashMap[Int, String] = new HashMap[Int,
String]()
```

Понятно, что было бы достаточно показать, что `x` относится к типу `HashMap` с `Int`-типами в качестве ключей и `String`-типами в качестве значений, только один раз, дважды повторять одно и то же нет смысла.

В Scala имеется весьма сложная система логического вывода типов, позволяющая опускать почти всю информацию о типах, которая обычно вызывает раздражение. В предыдущем примере вполне работоспособны и две менее раздражающие альтернативы:

```
val x = new HashMap[Int, String]()
val x: Map[Int, String] = new HashMap()
```

Вывод типа в Scala может заходить довольно далеко. Фактически пользовательский код нередко вообще обходится без явного задания типов. Поэтому программы на Scala часто выглядят похожими на программы, написанные на динамически типизированных языках сценариев. Это, в частности, справедливо для прикладного клиентского кода, склеиваемого из заранее написанных библиотечных компонентов. Но для самих библиотечных компонентов это менее характерно, поскольку в них зачастую применяются довольно сложные типы, не допускающие гибкого использования таких схем. И это вполне естественно. Ведь сигнатуры типов элементов, составляющих интерфейс многократно используемых компонентов, должны задаваться в явном виде, так как они составляют существенную часть соглашения между компонентом и его клиентами.

1.4. Истоки происхождения Scala

На замысел Scala повлияли многие языки программирования и идеи, выработанные на основе исследования этих языков. Фактически новыми в Scala являются лишь немногие особенности — большинство из них уже применялось в том или ином виде в других языках программирования. Инновации в Scala

появляются в основном из того, как его конструкции сводятся воедино. В этом разделе будут перечислены основные факторы, оказавшие влияние на конструкцию языка Scala. Перечень не может быть исчерпывающим, поскольку в конструкциях языков программирования так много толковых идей, что перечислить здесь их все просто невозможно.

На внешнем уровне Scala позаимствовал существенную часть синтаксиса у Java и C#, которые в свою очередь взяли большинство своих синтаксических соглашений у C и C++. Выражения, инструкции и блоки в основном из Java, как, собственно, и синтаксис классов, создание пакетов и импорт16. Кроме синтаксиса, Scala перенял и другие элементы Java: его основные типы, библиотеки классов и модель выполнения.

Scala многим обязан и другим языкам. Его однородная модель объектов впервые появилась в Smalltalk и впоследствии была принята языком Ruby. Идея универсальной вложенности (почти каждая конструкция в Scala может быть вложена внутрь любой другой конструкции) реализована и в Algol, Simula, а в последнее время в Beta и gbeta. Принцип универсального доступа к вызову методов и выбору полей пришел из Eiffel. Подход к функциональному программированию очень близок по духу к применяемому в семействе языков ML, видными представителями которого являются SML, OCaml и F#. Многие функции высшего порядка в стандартной библиотеке Scala присутствуют также в ML или Haskell. Толчком для появления в Scala подразумеваемых параметров стали классы типов языка Haskell — в более классическом объектно-ориентированном окружении они дают аналогичные результаты. Используемая в Scala основная библиотека многопоточного вычисления на основе акторов, Akka, создавалась под сильным влиянием особенностей языка Erlang.

Scala — не первый язык, в котором акцент сделан на масштабируемости и расширяемости. Исторические корни расширяемых языков для различных областей применения обнаруживаются в статье Петера Ландина (Peter Landin) 1966 года

The Next 700 Programming Languages («Следующие 700 языков программирования»)17. (Язык Iswim, описываемый в этой статье, наряду с Lisp является одним из первых в своем роде функциональных языков.) Происхождение конкретной идеи трактовки инфиксного оператора в качестве функции может быть прослежено до Iswim и Smalltalk. Еще одна важная мысль заключается в разрешении использования функционального литерала (или блока) в качестве параметра, позволяющего библиотекам определять структуры управления. Она также проистекает из Iswim и Smalltalk. Как у Smalltalk, так и у Lisp довольно гибкий синтаксис, широко применявшийся для создания внутренних предметно-ориентированных языков. Еще один масштабируемый язык, C++, который может быть адаптирован и расширен благодаря использованию перегрузки операторов и своей системе шаблонов, построен, по сравнению со Scala, на низкоуровневой и более системно-ориентированной основе. Scala также не первый язык, объединяющий в себе функциональное и объектно-ориентированное программирование, хотя, наверное, в этом направлении он продвинулся гораздо дальше прочих. К числу других языков, объединивших некоторые элементы функционального программирования в объектно-ориентированном программировании (object-oriented programming (OOP)), относятся Ruby, Smalltalk и Python. Расширения Java-подобного ядра некоторыми функциональными идеями были предприняты на Java-платформе в Pizza, Nice, Multi-Java и самом Java 8. Существуют также изначально функциональные языки, которые приобрели систему объектов. В качестве примера можно привести OCaml, F# и PLT-Scheme.

В Scala применяются также некоторые нововведения в области языков программирования. Например, его абстрактные типы — это более объектно-ориентированная альтернатива обобщенным типам, его трейты позволяют выполнять гибкую сборку компонентов, а экстракторы обеспечивают независимый от представления способ поиска по шаблону. Эти нововведения были

изложены в статьях на конференциях по языкам программирования в последние годы.

Резюме

Ознакомившись с этой главой, вы получили некоторое представление о том, что такое язык Scala и как он может помочь программисту в работе. Разумеется, Scala не решит все ваши проблемы и не повысит волшебным образом вашу личную продуктивность. Следует заранее предупредить, что Scala нужно применять искусно, а для этого потребуется приобретение некоторых знаний и практических навыков. Если вы пришли к Scala от языка Java, то одними из наиболее сложных аспектов его изучения для вас могут стать система типов Scala, которая существенно богаче, чем у Java, и его поддержка функционального стиля программирования. Цель данной книги — послужить руководством при поэтапном, от простого к сложному, изучении особенностей Scala. Полагаем, что вы приобретете весьма полезный опыт, расширяющий ваш кругозор и изменяющий взгляд на проектирование программных средств. Надеемся, что вы также получите от программирования на Scala истинное удовольствие и творческое вдохновение.

В следующей главе мы приступим к написанию кода на Scala.

1 Scala произносится как «скала».

2 Пожалуйста, не сердитесь на нас, если не сможете разобраться со всеми тонкостями этой программы. Объяснения будут даны в двух следующих главах.

3 *Raymond E.* The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. — O'Reilly, 1999.

4 *Steele Jr., Guy L.* Growing a Language. Higher-Order and Symbolic Computation, 12:221–223, 1999. Transcript of a talk given at OOPSLA, 1998.

5 factorial(x), или x! в математической записи, является результатом вычисления 1 × 2 × × … × x, где для 0! определено значение 1.

6 Scala поставляется со стандартной библиотекой, часть которой будет рассмотрена в данной книге. За дополнительной информацией можно обратиться к имеющейся в

библиотеке документации Scaladoc, доступной в дистрибутиве и в Интернете по адресу: http://www.scala-lang.org.

7 *Kay A. C.* The Early History of Smalltalk. In History of programming languages. — II, P. 511–598. — New York: ACM, 1996.

8 Начиная с Java 8, у интерфейсов могут быть исходные реализации методов, но они не предлагают всех тех свойств, что имеются у трейтов языка Scala.

9 Изначально существовала реализация Scala, запускаемая на платформе .NET, но она больше не используется. В последнее время все большую популярность набирает реализация Scala под названием Scala.js, запускаемая на JavaScript.

10 Единственное отличие заключается в том, что переменные экземпляра, полученные в случае применения Scala, будут финальными (final).

11 Функциональный литерал может называться предикатом, если типом его результата будет Boolean.

12 Такое использование символа подчеркивания в качестве заместителя для аргументов рассматривается в разделе 8.5.

13 Обобщенные типы рассматриваются в главе 19, пересечения (например, A с B с C) — в главе 12, а абстрактные типы — в главе 20.

14 *Kay A. C.* An email to Stefan Ram on the meaning of the term “object-oriented programming”. July 2003. The email is published on the web at. http://www.purl.org/stefan_ram/pub/doc_kay_oop_en (accessed June 6, 2008).

15 *Dijkstra E.W.* Notes on Structured Programming. April 1970. Circulated privately. Available at http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF as EWD249 (accessed June 6, 2008).

16 Главное отличие от Java касается синтаксиса для аннотаций типов: вместо «Тип переменная», как в Java, используется форма «переменная: Тип». Применяемый в Scala постфиксный синтаксис типа похож на синтаксис Pascal, Modula-2 или Eiffel. Основная причина такого отклонения имеет отношение к логическому выводу типа, зачастую позволяющему опускать тип переменной или тип возвращаемого методом значения. Легче использовать синтаксис «переменная: Тип», поскольку двоеточие и тип можно просто не указывать. Но в стиле языка C, использующего форму «Тип переменная», просто так не указывать тип нельзя, поскольку при этом исчезнет сам признак начала определения. Для неуказанного типа в качестве заполнителя требуется какое-нибудь ключевое слово (C# 3.0, в котором имеется логический вывод типа, для этой цели использует ключевое слово var). Такое альтернативное ключевое слово представляется несколько более надуманным и менее привычным, чем подход, используемый в Scala.

17 *Landin P. J.* The Next 700 Programming Languages // Communications of the ACM. — 1966. — 9(3):157–166.

2. Первые шаги в Scala

Настало время написать какой-нибудь код на языке Scala. Прежде чем перейти к углубленному изучению Scala, прочтите две вводные главы и, что наиболее важно, попробуйте самостоятельно написать код. Рекомендуем по мере освоения материала на практике проверить работу всех примеров кода, представленных в этой и последующих главах. Лучше всего приступить к изучению Scala, программируя на этом языке.

Для запуска представленных далее примеров у вас должна быть стандартная установка Scala. Получить ее можно по адресу **http://www.scala-lang.org/downloads**, следуя инструкциям для вашей платформы. Вы можете воспользоваться также дополнительным модулем Scala для Eclipse, IntelliJ или NetBeans. Применительно к шагам, рассматриваемым в данной главе, предполагаем, что вы воспользовались дистрибутивом Scala из **scala-lang.org**[18].

Если вы опытный программист, но новичок в Scala, следующие две главы дадут вам достаточный объем знаний, позволяющий приступить к написанию полезных программ на Scala. Если опыт программирования у вас невелик, то часть материала может показаться чем-то загадочным. Но не стоит переживать. Для ускорения процесса изучения нам пришлось обойтись без некоторых подробностей. Более обстоятельно все будет объяснено далее. Кроме того, в следующих двух главах дается ряд сносок, где указаны разделы книги, в которых можно найти более расширенную информацию.

Шаг 1. Освоение интерпретатора Scala

Проще всего приступить к изучению Scala путем использования Scala-интерпретатора — интерактивной оболочки для написания выражений и программ на Scala. Интерпретатор, который называется *scala*, будет вычислять набираемые вами выражения и

выводить на экран получающееся значение. Чтобы воспользоваться интерпретатором, нужно в приглашении командной строки набрать команду `scala`[19]:

```
$ scala
Welcome to Scala version 2.11.7
Type in expressions to have them evaluated.
Type :help for more information.
scala>
```

После набора выражения, например `1 + 2`, и нажатия клавиши **Enter**:

```
scala> 1 + 2
```

интерпретатор выведет на экран:

```
res0: Int = 3
```

Эта строка включает:

- автоматически сгенерированное или определенное пользователем имя для ссылки на вычисленное значение (`res0`, означающее результат 0);
- двоеточие (`:`), за которым следует тип выражения (`Int`);
- знак равенства (`=`);
- значение, полученное в результате вычисления выражения (`3`).

Тип `Int` указывает на класс `Int` в пакете `scala`. Пакеты в Scala аналогичны пакетам в Java — они разбивают глобальное пространство имен на части и предоставляют механизм для сокрытия данных[20]. Значения класса `Int` соответствуют `int`-значениям в Java. Если говорить в общем, то у всех элементарных типов Java есть конкретные классы в пакете `scala`. Например,

scala.Boolean соответствует Java-типу boolean. А scala.Float соответствует Java-типу float. И при компиляции вашего кода Scala в байт-код Java компилятор Scala будет использовать элементарные типы Java там, где возможно обеспечить вам преимущества в производительности при работе с элементарными типами.

Идентификатор resX может использоваться в последующих строках. Например, поскольку ранее для res0 было установлено значение 3, то выражение res0 * 3 будет вычислено в 9:

```
scala> res0 * 3
res1: Int = 9
```

Для вывода на экран необходимого, но недостаточно информативного приветствия Hello, world! наберите следующую команду:

```
scala> println("Hello, world!")
Hello, world!
```

Функция println выводит на стандартное устройство вывода переданную ей строку подобно тому, как это делает toSystem.out.println в Java.

Шаг 2. Определение переменных

В Scala имеются две разновидности переменных: val-переменные и var-переменные. Val-переменные аналогичны финальным переменным в Java. После инициализации val-переменная уже никогда не может быть повторно присвоена. В отличие от нее, var-переменная аналогична нефинальной переменной в Java. В течение своего жизненного цикла var-переменная может быть присвоена повторно. Определение val-переменной выглядит следующим образом:

```
scala> val msg = "Hello, world!"
msg: String = Hello, world!
```

Эта инструкция вводит в употребление переменную `msg` в качестве имени для строки `"Hello, world!"`. Типом `msg` является `java.lang.String`, поскольку строки в Scala реализуются Java-классом `String`.

Если вы привыкли объявлять переменные в Java, то в этом примере кода можете заметить одно существенное отличие: в `val`-определении нигде не фигурируют ни `java.lang.String`, ни `String`. Этот пример демонстрирует *логический вывод типа*, то есть возможность Scala определять неуказанные типы. В данном случае, поскольку вы инициализировали `msg` строковым литералом, Scala придет к выводу, что типом `msg` должен быть `String`. Когда интерпретатор (или компилятор) Scala хочет выполнить вывод типа, зачастую лучше всего будет позволить ему сделать это, не засоряя код ненужной явной аннотацией типа. Но если хотите, можете указать тип явно, и, наверное, иногда это придется делать. Явная аннотация типа может не только гарантировать, что компилятор Scala выведет желаемый тип, но и послужить полезной документацией для тех, кто впоследствии станет читать ваш код. В отличие от Java, где тип переменной указывается перед ее именем, в Scala вы задаете тип переменной после ее имени, отделяя его двоеточием, например:

```
scala> val msg2: java.lang.String = "Hello again,
world!"
msg2: String = Hello again, world!
```

Или же, поскольку типы `java.lang` опознаются в программах на Scala по их простым именам[21], запись можно упростить:

```
scala> val msg3: String = "Hello yet again,
world!"
msg3: String = Hello yet again, world!
```

Вернемся к исходной переменной msg. Поскольку она определена, ею можно воспользоваться в соответствии с вашими ожиданиями, например:

```
scala> println(msg)
Hello, world!
```

Учитывая, что msg является val-, а не var-переменной, вы не сможете повторно присвоить ей другое значение22. Посмотрите, к примеру, как интерпретатор выражает свое недовольство при попытке сделать следующее:

```
scala> msg = "Goodbye cruel world!"
<console>:8: error: reassignment to val
       msg = "Goodbye cruel world!"
           ^
```

Если нужно выполнить повторное присваивание, следует воспользоваться var-переменной:

```
scala> var greeting = "Hello, world!"
greeting: String = Hello, world!
```

Как только приветствие станет var-, а не val-переменной, ему можно будет присвоить другое значение. Если, к примеру, вы станете раздражительнее, можете поменять приветствие на просьбу оставить вас в покое:

```
scala> greeting = "Leave me alone, world!"
greeting: String = Leave me alone, world!
```

Для ввода в интерпретатор кода, не помещающегося в одну строку, просто продолжайте набирать код после заполнения первой строки. Если набор кода еще не завершен, интерпретатор отреагирует установкой на следующей строке вертикальной черты:

```
scala> val multiLine =
     | "This is the next line."
multiLine: String = This is the next line.
```

Если поймете, что набрали что-то не то, а интерпретатор все еще ждет ввода дополнительного текста, можете сделать отмену, дважды нажав клавишу **Enter**:

```
scala> val oops =
     |
     |
You typed two blank lines. Starting a new command.
scala>
```

Далее в книге символы вертикальной черты отображаться не будут, чтобы код легче читался и его было проще скопировать и вставить в интерпретатор из электронной версии в формате PDF.

Шаг 3. Определение функций

После работы с переменными в Scala вам, наверное, захотелось написать какие-нибудь функции. В Scala это делается так:

```
scala> def max(x: Int, y: Int): Int = {
         if (x > y) x
         else y
       }
max: (x: Int, y: Int)Int
```

Определение функции начинается с ключевого слова `def`. После имени функции, в данном случае `max`, стоит заключенный в круглые скобки перечень параметров, разделенных запятыми. За каждым параметром функции должна следовать аннотация типа, перед которой ставится двоеточие, поскольку компилятор Scala (и

интерпретатор, но с этого момента будет упоминаться только компилятор) не выводит типы параметров функции. В данном примере функция по имени `max` получает два параметра, `x` и `y`, и оба они относятся к типу `Int`. После закрывающей круглой скобки перечня параметров функции `max` обнаруживается аннотация типа `: Int`. Она определяет *тип результата* самой функции `max`[23]. За типом результата функции стоят знак равенства и пара фигурных скобок, внутри которых расположено тело функции.

В данном случае в теле содержится одно выражение `if`, с помощью которого в качестве результата выполнения функции `max` выбирается либо `x`, либо `y` в зависимости от того, значение какой из переменных больше. Как здесь показано, выражение `if` в Scala может вычисляться в значение, подобное тому, что вычисляется тернарным оператором Java. Например, Scala-выражение `if (x > y) x else y` вычисляется точно так же, как выражение `(x > y) ? x : y` в Java. Знак равенства, предшествующий телу функции, дает понять, что с точки зрения функционального мира функция определяет выражение, результатом вычисления которого становится значение. Основная структура функции показана на рис. 2.1.

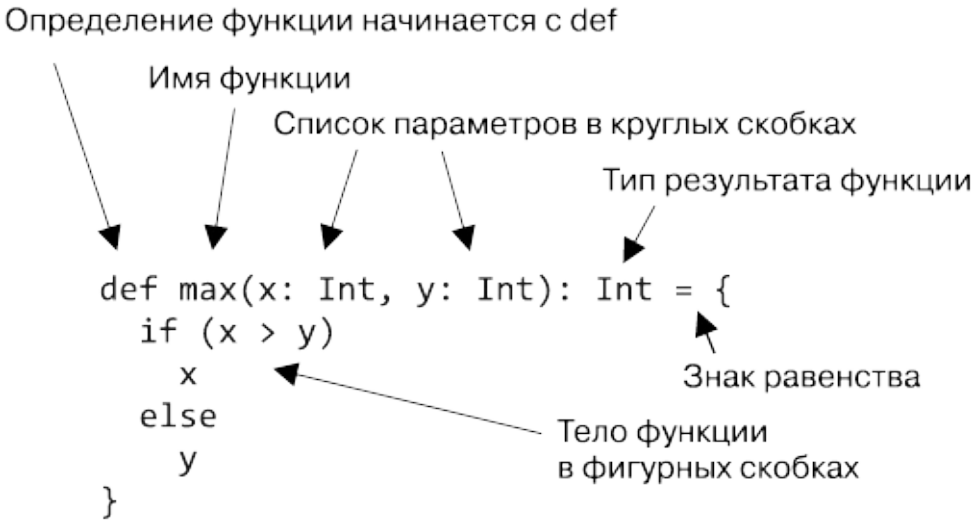


Рис. 2.1. Основная форма определения функции в Scala

Иногда компилятор Scala может потребовать от вас указать тип результата функции. Если, к примеру, функция является *рекурсивной*[24], вы должны явно задать тип ее результата. Но в случае с функцией `max` вы можете не указывать тип результата функции — компилятор выведет его самостоятельно[25]. Если функция состоит только из одной инструкции, то при желании фигурные скобки можно не ставить. Следовательно, альтернативный вариант функции `max` может быть таким:

```
scala> def max(x: Int, y: Int) = if (x > y) x else
y
max: (x: Int, y: Int)Int
```

После определения функции ее можно вызвать по имени:

```
scala> max(3, 5)
res4: Int = 5
```

А вот определение функции, не получающей параметров и не возвращающей полезного результата:

```
scala> def greet() = println("Hello, world!")
greet: ()Unit
```

Когда определяется функция приветствия `greet()`, интерпретатор откликается следующим приветствием: `()Unit`. Разумеется, слово greet — это имя функции. Пустота в скобках показывает, что функция не получает параметров. А `Unit` является типом результата функции `greet`. Тип результата `Unit` показывает, что функция не возвращает никакого интересного значения. Тип `Unit` в Scala подобен типу `void` в Java. Фактически каждый метод, возвращающий `void` в Java, отображается на метод, возвращающий `Unit` в Scala. Таким образом, методы с типом результата `Unit` выполняются только для того, чтобы проявились их побочные эффекты. В случае с `greet()` побочным эффектом

будет дружеское приветствие, выведенное на устройство стандартного вывода.

При выполнении следующего шага код Scala будет помещен в файл и запущен в качестве сценария. Если нужно выйти из интерпретатора, это можно сделать путем ввода команды `:quit` или `:q`:

```
scala> :quit
$
```

Шаг 4. Создание сценариев на языке Scala

Несмотря на то что язык Scala разработан, чтобы помочь программистам в создании очень больших масштабируемых систем, он вполне может подойти и для решения менее масштабных задач наподобие написания сценариев. Сценарий представляет собой простую последовательность инструкций, размещенных в файле и выполняемых друг за другом. Поместите в файл по имени `hello.scala` следующий код:

```
println("Hello, world, from a script!")
```

а затем запустите файл на выполнение[26]:

```
$ scala hello.scala
```

И вы получите еще одно приветствие:

```
Hello, world, from a script!
```

Аргументы командной строки, указанные для сценария Scala, можно получить из Scala-массива по имени `args`. В Scala индексация элементов массива начинается с нуля и обращение к элементу выполняется указанием индекса в круглых скобках. Следовательно, первым элементом в Scala-массиве по имени `steps` будет `steps(0)`, а не `steps[0]`, как в Java. Убедиться в этом

на практике можно, создав новый файл по имени `helloarg.scala`:

```
// Поприветствуйте содержимое первого аргумента
println("Hello, " + args(0) + "!")
```

а затем запустите его на выполнение:

```
$ scala helloarg.scala planet
```

В этой команде `planet` передается в качестве аргумента командной строки, доступного в сценарии при использовании выражения `args(0)`. Поэтому вы должны увидеть на экране следующий текст:

```
Hello, planet!
```

Обратите внимание на наличие комментария в сценарии. Компилятор Scala проигнорирует символы между `//` и концом строки, а также все символы между сочетаниями символов `/*` и `*/`. В этом примере также показано объединение `String`-значений, выполненное с помощью оператора `+`. Весь код работает вполне предсказуемо. Выражение `"Hello, " + "world!"` будет вычислено в строку `"Hello, world!"`.

Шаг 5. Организация цикла с помощью while и принятие решения с помощью if

Чтобы попробовать в работе конструкцию `while`, наберите следующий код и сохраните его в файле `printargs.scala`:

```
var i = 0
while (i < args.length) {
  println(args(i))
  i += 1
```

```
}
```

ПРИМЕЧАНИЕ

Хотя примеры в данном разделе помогают объяснить суть циклов while, они не демонстрируют наилучший стиль программирования на Scala. В следующем разделе будут показаны более рациональные подходы, позволяющие избежать перебора элементов массива с помощью индексов.

Этот сценарий начинается с определения переменной `var i = 0`. Вывод типа относит переменную `i` к типу `scala.Int`, поскольку это тип ее начального значения `0`. Конструкция `while` на следующей строке заставляет блок (код между фигурными скобками) повторно выполняться, пока булево выражение `i < args.length` будет вычисляться в `false`. Метод `args.length` рассчитывает длину массива `args`. Блок содержит две инструкции, каждая из которых набрана с отступом в два пробела, что является рекомендуемым стилем отступов для кода на Scala. Первая инструкция, `println(args(i))`, выводит на экран `i`-й аргумент командной строки. Вторая инструкция, `i += 1`, увеличивает значение переменной `i` на единицу. Следует заметить, что Java-код `++i` и `i++` в Scala не работает. Чтобы в Scala увеличить значение переменной на единицу, нужно использовать одно из двух выражений: либо `i = i + 1`, либо `i += 1`. Запустите этот сценарий с помощью следующей команды:

```
$ scala printargs.scala Scala is fun
```

и вы увидите:

```
Scala
is
fun
```

Чтобы развлекаться дальше, наберите в новом файле `echoargs.scala` следующий код:

```
var i = 0
while (i < args.length) {
  if (i != 0)
    print(" ")
  print(args(i))
  i += 1
}
println()
```

Для вывода всех аргументов в одной и той же строке в этой версии вместо вызова `println` используется вызов `print`. Чтобы эту строку можно было прочитать, перед каждым аргументом, за исключением первого, благодаря использованию конструкции `if (i != 0)` вставляется пробел. Поскольку при первом проходе цикла `while` выражение `i != 0` будет вычисляться в `false`, перед начальным элементом пробел выводиться не будет. В самом конце добавлена еще одна инструкция `println`, чтобы после вывода аргументов произошел переход на новую строку. Тогда у вас получится очень красивая картинка. Если запустить этот сценарий с помощью команды:

```
$ scala echoargs.scala Scala is even more fun
```

вы получите на экране такой текст:

```
Scala is even more fun
```

Обратите внимание, что в Scala, как и в Java, булевы выражения для `while` или `if` нужно помещать в круглые скобки. (Иными словами, вы не можете в Scala воспользоваться таким выражением, как `if i < 10`, вполне допустимым в таких языках, как Ruby. В Scala нужно воспользоваться записью `if (i < 10)`.) Еще один аналогичный применяемому в Java прием программирования

заключается в том, что, если в блоке `if` используется только одна инструкция, при желании фигурные скобки можно не ставить, как показано в конструкции `if` в файле `echoargs.scala`. Но хотя их там нет, в Scala, как и в Java, для отделения инструкций друг от друга используются точки с запятыми (стоит уточнить: в Scala точки с запятыми зачастую необязательны, что снижает нагрузку на правый мизинец при наборе текста). Но если вы склонны к многословию, то можете записать содержимое файла `echoargs.scala` в следующем виде:

```
var i = 0;
while (i < args.length) {
    if (i != 0) {
        print(" ");
    }
    print(args(i));
    i += 1;
}
println();
```

Шаг 6. Последовательный перебор элементов с помощью foreach и for

Возможно, при написании циклов `while` в предыдущем шаге вы даже не осознавали того, что программирование велось в императивном стиле. При работе в этом стиле, который обычно применяется с такими языками, как Java, C++ и C, императивные команды при последовательном переборе элементов в цикле выдаются поочередно и зачастую изменяемое состояние совместно используется различными функциями. Scala позволяет программировать в императивном стиле, но, узнав этот язык получше, вы, скорее всего, перейдете преимущественно на функциональный стиль. По сути, одной из целей этой книги

является помощь в освоении работы в функциональном стиле, чтобы она стала такой же комфортной, как и работа в императивном стиле.

Одной из основных характеристик функционального языка является то, что его функции относятся к конструкциям первого класса, и это абсолютно справедливо для языка Scala. Например, еще один гораздо более лаконичный способ вывода каждого аргумента командной строки выглядит следующим образом:

```
args.foreach(arg => println(arg))
```

В этом коде в отношении массива `args` вызывается метод `foreach`, передающий результат своей работы в функцию. В данном случае передача выполняется в *функциональный литерал*, получающий один параметр по имени `arg`. Телом функции является вызов `println(arg)`. Если набрать показанный ранее код в новом файле по имени `pa.scala` и запустить этот файл на выполнение с помощью команды

```
$ scala pa.scala Concise is nice
```

то на экране появятся строки:

```
Concise
is
nice
```

В предыдущем примере интерпретатор Scala вывел тип `arg`, причислив эту переменную к `String`, поскольку `String` является типом элемента массива, в отношении которого вызван метод `foreach`. Если вы предпочитаете конкретизировать, можете упомянуть название типа. Но если пойти по этому пути, придется заключать ту часть кода, где указывается переменная аргумента, в круглые скобки, что является обычной формой повсеместно применяемого синтаксиса:

```
args.foreach((arg: String) => println(arg))
```

При запуске этот сценарий ведет себя точно так же, как и предыдущий.

Если же вы склонны не к конкретизации, а к более лаконичному изложению кода, можете воспользоваться специальными сокращениями, принятыми в Scala. Если функциональный литерал функции состоит из одной инструкции, принимающей один аргумент, обозначать этот аргумент явным образом по имени не нужно[27]. Поэтому работать будет и следующий код:

```
args.foreach(println)
```

Резюмируем усвоенное: синтаксис для функционального литерала представляет собой список поименованных параметров, заключенный в круглые скобки, а также правую стрелку, за которой следует тело функции. Этот синтаксис показан на рис. 2.2.

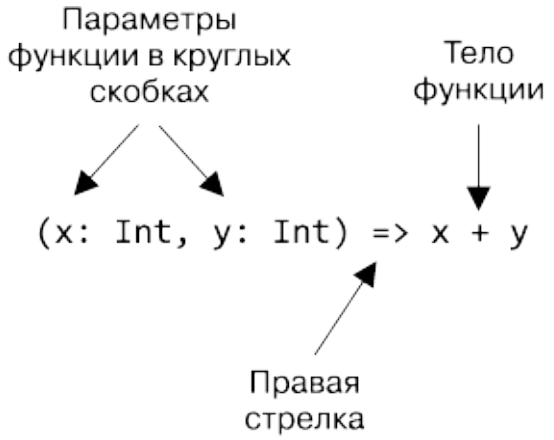


Рис. 2.2. Синтаксис функционального литерала в Scala

Теперь вы можете поинтересоваться: что же случилось с теми проверенными циклами `for`, которые вы привыкли использовать в таких императивных языках, как Java или C? Чтобы придерживаться функционального направления, в Scala есть только один функциональный родственник императивной конструкции `for`, который называется *выражением for*. Так как вы

не сможете понять всю его эффективность и выразительность, пока не доберетесь до раздела 7.3 (или не заглянете в него), здесь о нем будет дано только общее представление. Наберите в новом файле по имени `forargs.scala` следующий код:

```
for (arg <- args)
println(arg)
```

В круглых скобках после `for` содержится выражение `arg <- args`[28]. Справа от обозначения `<-` указан уже знакомый вам массив `args`. Слева от `<-` указана переменная `arg`, относящаяся к `val`-, а не к `var`-переменным. (Поскольку она всегда относится к `val`-переменным, то записывается только `arg`, а не `val arg`.) Хотя может показаться, что `arg` связана с `var`-переменной, поскольку она будет получать новое значение при каждой итерации, в действительности она относится к `val`-переменной: `arg` не может получить новое значение внутри тела выражения. Вместо этого для каждого элемента массива `args` будет создана новая `val`-переменная по имени `arg`, которая инициализируется значением элемента, и тело `for` будет выполнено.

Если сценарий `forargs.scala` запустить с помощью команды

```
$ scala forargs.scala for arg in args
```

то вы увидите:

```
for
arg
in
args
```

Диапазон применения используемого в Scala выражения `for` значительно шире, чем показано здесь, но для начала достаточно и этого примера. Дополнительные возможности `for` будут

рассмотрены в разделе 7.3 и в главе 23.

Резюме

Во второй главе вы прочли об основных принципах Scala и, надеемся, уже попробовали написать код на этом языке. В следующей главе мы продолжим вводный обзор и перейдем к изучению более сложных тем.

18 Приводимые в книге примеры тестировались с применением Scala версии 2.11.7.

19 Если вы используете Windows, то команду scala нужно набирать в окне командной строки.

20 Если вы не знакомы с пакетами Java, их можно рассматривать как средство предоставления классам полных имен. Поскольку Int входит в пакет scala, то Int является простым именем класса, а scala.Int — его полным именем. Более подробно пакеты рассматриваются в главе 13.

21 Простым именем java.lang.String является String.

22 Но в интерпретаторе новую val-переменную можно определить с именем, которое до этого уже использовалось. Такой механизм рассматривается в разделе 7.7.

23 В Java тип возвращаемого из метода значения является возвращаемым типом. В Scala то же самое понятие называется типом результата.

24 Функция называется рекурсивной, если она вызывает саму себя.

25 Тем не менее зачастую есть смысл показывать тип результата явно, даже когда компилятор этого не требует. Такая аннотация типа может упростить чтение кода, поскольку читателю не придется изучать тело функции для определения того, каким будет вывод типа результата.

26 В Unix и Windows сценарии можно запускать, не набирая слова scala, а используя синтаксис «решетка — восклицательный знак», показанный в приложении.

27 Это сокращение, называемое частично применяемой функцией, рассматривается в разделе 8.6.

28 Обозначение <– можно трактовать как «в». То есть код for (arg <– args) может быть прочитан как «для arg в args».

3. Последующие шаги в Scala

В этой главе мы продолжим знакомство с языком Scala. Здесь будут рассмотрены более сложные функциональные возможности. Когда вы усвоите материал главы, у вас будет достаточно знаний для начала создания полезных сценариев на Scala. Мы вновь рекомендуем по мере чтения текста получать практические навыки с помощью приводимых примеров. Лучше всего осваивать Scala, начиная создавать код на этом языке.

Шаг 7. Параметризация массивов типами

В Scala создавать объекты или экземпляры класса можно с помощью ключевого слова `new`. При создании объекта в Scala вы можете выполнить его параметризацию с использованием значений и типов. Параметризация означает конфигурирование экземпляра при его создании. Параметризация экземпляра значениями производится путем передачи конструктору объектов в круглых скобках. Например, следующий код Scala:

```
val big = new java.math.BigInteger("12345")
```

создает новый объект `java.math.BigInteger`, выполняя его параметризацию значением `"12345"`.

Параметризация экземпляра типами производится указанием одного или нескольких типов в квадратных скобках. Пример показан в листинге 3.1. Здесь `greetStrings` является типом `Array[String]` («массив строк»), инициализируемым длиной `3` путем его параметризации значением `3` в первой строке кода. Если запустить код в листинге 3.1 в качестве сценария, вы увидите еще одно приветствие `Hello, world!`. Учтите, что при параметризации экземпляра как типом, так и значением тип стоит первым и указывается в квадратных скобках, а за ним следует

значение в круглых скобках.

Листинг 3.1. Параметризация массива типом

```
val greetStrings = new Array[String](3)

greetStrings(0) = "Hello"
greetStrings(1) = ", "
greetStrings(2) = "world!\n"

for (i <- 0 to 2)
  print(greetStrings(i))
```

ПРИМЕЧАНИЕ

Хотя код в листинге 3.1 содержит важные понятия, он не показывает рекомендуемый способ создания и инициализации массива в Scala. Более рациональный способ будет показан в листинге 3.2.

Если вы склонны делать более явные указания, тип `greetStrings` можно обозначить так:

```
val greetStrings: Array[String] = new Array[String](3)
```

При условии применения имеющегося в Scala вывода типов эта строка кода семантически эквивалентна первой строке листинга 3.1. Но в этой форме показано, что, хотя та часть параметризации, которая относится к типу (название типа в квадратных скобках), формирует часть типа экземпляра, часть параметризации, которая относится к значению (значения в круглых скобках), в формировании не участвует. Типом

greetStrings является Array[String], а не Array[String](3).

В следующих трех строках кода в листинге 3.1 инициализируется каждый элемент массива greetStrings:

```
greetStrings(0) = "Hello"
greetStrings(1) = ", "
greetStrings(2) = "world!\n"
```

Как уже упоминалось, доступ к массивам в Scala осуществляется за счет помещения индекса элемента в круглые, а не квадратные скобки, как в Java. Следовательно, нулевым элементом массива будет greetStrings(0), а не greetStrings[0].

Эти три строки кода иллюстрируют важное понятие, помогающее осмыслить значение для Scala val-переменных. Когда переменная определяется с помощью val, повторно присвоить ей значение нельзя, но объект, на который она ссылается, потенциально может быть изменен. Следовательно, в данном случае задать greetStrings значение другого массива невозможно — переменная greetStrings всегда будет указывать на один и тот же экземпляр типа Array[String], с которым она была инициализирована. Но впоследствии в элементы типа Array[String] можно вносить изменения, следовательно, сам массив является изменяемым.

Последние две строки листинга 3.1 содержат выражение for, которое поочередно выводит каждый элемент greetStringsarray:

```
for (i <- 0 to 2)
  print(greetStrings(i))
```

В первой строке кода для этого выражения for показано еще одно общее правило Scala: если метод получает только один параметр, его можно вызвать без точки или круглых скобок. В этом примере to на самом деле является методом, получающим один

Int-аргумент. Код `0 to 2` преобразуется в вызов метода `(0).to(2)`[29]. Следует заметить, что этот синтаксис работает только при явном указании получателя вызова метода. Использовать код `println 10` нельзя, а код `Console println 10` — можно.

С технической точки зрения в Scala нет перегрузки операторов, поскольку в нем фактически нет операторов в традиционном понимании. Вместо этого такие символы, как `+`, `-`, `*`, `/`, могут использоваться в качестве имен методов. Следовательно, когда при выполнении шага 1 вы набираете в интерпретаторе Scala код `1 + 2`, в действительности вы вызываете метод по имени `+` в отношении Int-объекта `1`, передавая ему в качестве параметра значение `2`. Как показано на рис. 3.1, вместо этого `1 + 2` можно записать с использованием традиционного синтаксиса вызова метода: `(1).+(2)`.

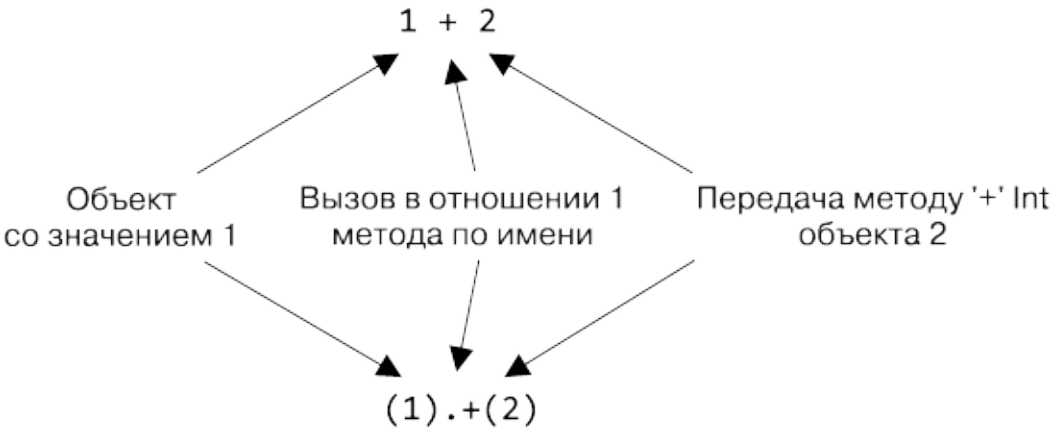


Рис. 3.1. Все операции в Scala являются вызовами методов

Еще одна весьма важная идея, проиллюстрированная в этом примере, поможет понять, почему доступ к элементам массивов Scala осуществляется с использованием круглых скобок. В Scala меньше особых случаев по сравнению с Java. Массивы в Scala, как и в случае с любыми другими классами, являются просто экземплярами классов. При использовании круглых скобок, окружающих одно или несколько значений переменной, Scala

преобразует код в вызов метода по имени `apply` применительно к этой переменной. Следовательно, код `greetStrings(i)` преобразуется в код `greetStrings.apply(i)`. Получается, что элемент массива в Scala является просто вызовом обычного метода, ничем не отличающегося от любого своего собрата. Этот принцип не ограничивается массивами: любое применение объекта в отношении каких-либо аргументов в круглых скобках будет преобразовано в вызов метода `apply`. Разумеется, этот код будет откомпилирован, только если в этом типе объекта определяется метод `apply`. То есть это не особый случай, а общее правило.

По аналогии с этим, когда присваивание выполняется в отношении переменной, к которой применены круглые скобки с одним или несколькими аргументами внутри, компилятор выполнит преобразование в вызов метода `update`, получающего не только аргументы в круглых скобках, но и объект, расположенный справа от знака равенства. Например, код:

```
greetStrings(0) = "Hello"
```

будет преобразован в код:

```
greetStrings.update(0, "Hello")
```

Таким образом, следующий код является семантическим эквивалентом коду листинга 3.1:

```
val greetStrings = new Array[String](3)
greetStrings.update(0, "Hello")
greetStrings.update(1, ", ")
greetStrings.update(2, "world!\n")
for (i <- 0.to(2))
  print(greetStrings.apply(i))
```

Концептуальная простота в Scala достигается за счет того, что

всё, от массивов до выражений, рассматривается как объекты с методами. Вам не нужно запоминать особые случаи, например такие, как существующее в Java различие между элементарными типами и соответствующими им типами-оболочками или между массивами и обычными объектами. Более того, такое однообразие не вызывает больших издержек производительности. Компилятор Scala везде, где только возможно, использует в откомпилированном коде массивы Java, элементарные типы и чистые арифметические операции.

Хотя рассмотренные до сих пор в этом шаге примеры компилируются и выполняются весьма неплохо, в Scala имеется более лаконичный способ создания и инициализации массивов, который, как правило, вы и будете использовать (листинг 3.2). Этот код создает новый массив длиной в три элемента, инициализируемый переданными строками `"zero"`, `"one"` и `"two"`. Компилятор выводит тип массива как `Array[String]`, поскольку ему передаются строки.

Листинг 3.2. Создание и инициализация массива

```
val numNames = Array("zero", "one", "two")
```

Фактически в листинге 3.2 вызывается фабричный метод по имени `apply`, создающий и возвращающий новый массив. Этот метод `apply` получает переменное количество аргументов[30] и определяется в объекте-спутнике `Array`. Подробнее объекты-спутники будут рассматриваться в разделе 4.3. Если вам приходилось программировать на Java, можете воспринимать это как вызов статического метода по имени `apply` в отношении класса `Array`. Менее лаконичный способ вызова того же метода `apply` выглядит так:

```
val numNames2 = Array.apply("zero", "one", "two")
```

Шаг 8. Использование списков

Одной из превосходных отличительных черт функционального стиля программирования является полное отсутствие у методов побочных эффектов. Единственным действием метода должно быть вычисление и возвращение значения. Получаемые от применения такого подхода преимущества заключаются в том, что методы становятся менее запутанными, что упрощает их чтение и повторное использование. Еще одно преимущество (в статически типизированных языках) заключается в том, что все, попадающее в метод и выходящее за его пределы, проходит проверку на принадлежность к определенному типу, поэтому логические ошибки, скорее всего, проявятся сами по себе в виде ошибок типов. Применение данной функциональной философии к миру объектов означает превращение этих объектов в неизменяемые.

Как вы уже видели, массив Scala является неизменяемой последовательностью объектов с общим типом. Тип `Array[String]`, к примеру, содержит только строки. Изменить длину массива после создания его экземпляра невозможно, но вы можете менять значения его элементов. Таким образом, массивы относятся к изменяемым объектам.

Для неизменяемой последовательности объектов с общим типом можно воспользоваться списком, определяемым Scala-классом `List`. Как и в случае массивов, в типе `List[String]` содержатся только строки. Список Scala, `scala.List`, отличается от Java-типа `java.util.List` тем, что списки Scala всегда неизменямые, а списки Java могут изменяться. В более общем смысле список Scala разработан с прицелом на применение функционального стиля программирования. Список создается очень просто, и листинг 3.3 как раз это и показывает.

Листинг 3.3. Создание и инициализация списка

```
val oneTwoThree = List(1, 2, 3)
```

Код в листинге 3.3 создает новую val-переменную по имени oneTwoThree, инициализируемую типом newList[Int] с целочисленными элементами 1, 2 и 3[31]. Из-за своей неизменяемости списки ведут себя подобно строкам в Java: при вызове метода в отношении списка из-за имени этого метода может казаться, что обрабатываемый список будет изменен, но вместо этого создается и возвращается новый список с новым значением. Например, в List для объединения списков имеется метод, обозначаемый как :::. Используется он следующим образом:

```
val oneTwo = List(1, 2)
val threeFour = List(3, 4)
val oneTwoThreeFour = oneTwo ::: threeFour
println(oneTwo + " and " + threeFour + " were not
mutated.")
println("Thus, " + oneTwoThreeFour + " is a new
list.")
```

Запустив этот сценарий, вы увидите следующую картину:

```
List(1, 2) and List(3, 4) were not mutated.
Thus, List(1, 2, 3, 4) is a new list.
```

Возможно, со списками чаще всего будет использоваться оператор ::, который произносится cons («конс»). Он добавляет в начало существующего списка новый элемент и возвращает получившийся в результате этого список. Например, если запустить на выполнение следующий сценарий:

```
val twoThree = List(2, 3)
val oneTwoThree = 1 :: twoThree
println(oneTwoThree)
```

вы увидите:

```
List(1, 2, 3)
```

ПРИМЕЧАНИЕ

В выражении 1 :: twoThree метод :: является методом его правого операнда — списка twoThree. Можно подумать, что с ассоциативностью метода :: что-то не то, но есть простое мнемоническое правило: если метод используется в виде оператора, например a * b, то он вызывается в отношении левого операнда, как в выражении a.*(b), если только имя метода не заканчивается двоеточием. А если оно заканчивается двоеточием, то метод вызывается в отношении правого операнда. Поэтому в выражении 1 :: twoThree метод :: вызывается в отношении twoThree с передачей ему 1, как здесь: twoThree.::(1). Ассоциативность операторов более подробно будет рассматриваться в разделе 5.9.

Исходя из того, что самым кратким вариантом указания пустого списка является `Nil`, одним из способов инициализации новых списков выступает связывание элементов с помощью `cons`-оператора с `Nil` в качестве последнего элемента[32]. Например, следующий сценарий создаст ту же картину на выходе, что и предыдущий `List(1, 2, 3)`:

```
val oneTwoThree = 1 :: 2 :: 3 :: Nil
println(oneTwoThree)
```

Имеющийся в Scala класс `List` укомплектован весьма полезными методами, многие из которых показаны в табл. 3.1. Вся эффективность списков будет раскрыта в главе 16.

Почему со списками не следует применять операцию добавления

В классе `List` предлагается операция добавления, которая

записывается как `:+` (о ней говорится в главе 24), но очень редко используется, поскольку время, которое она тратит на добавление к списку, увеличивается линейно с размером списка, а на добавление в начало списка с помощью метода `::` всегда затрачивается одно и то же время. Если нужно добиться эффективности при создании списка путем дополнения элементами, можно добавлять их в начало, а завершив добавление, вызвать метод реверсирования `reverse`. Или же можно воспользоваться изменяемым списком `ListBuffer`, предлагающим операцию добавления, а затем, завершив добавление, вызвать метод `toList` и преобразовать его в обычный список. Список типа `ListBuffer` будет рассмотрен в разделе 22.2.

Таблица 3.1. Некоторые методы класса List и их использование

Что используется	Что этот метод делает
List() или Nil	Создает пустой список List
List("Cool", "tools", "rule")	Создает новый список типа List[String] с тремя значениями: "Cool", "tools" и "rule"
val thrill = "Will" :: "fill" :: "until" :: Nil	Создает новый список типа List[String] с тремя значениями: "Will", "fill" и "until"
List("a", "b") ::: List("c", "d")	Объединяет два списка (возвращает новый список типа List[String] со значениями "a", "b", "c" и "d")
thrill(2)	Возвращает элемент с индексом 2 (при начале отсчета с нуля) списка thrill (возвращает "until")
thrill.count(s => s.length == 4)	Подсчитывает количество строковых элементов в thrill, имеющих длину 4 (возвращает 2)
thrill.drop(2)	Возвращает список thrill без его первых двух элементов (возвращает List("until"))
thrill.dropRight(2)	Возвращает список thrill без самых правых двух элементов (возвращает List("Will"))
thrill.exists(s => s ==	Определяет наличие в списке thrill строкового элемента, имеющего

"until")	значение "until" (возвращает true)
thrill.filter(s => s.length == 4)	Возвращает список всех элементов списка thrill, имеющих длину 4, соблюдая порядок их следования в списке (возвращает List("Will", "fill"))
thrill.forall(s => s.endsWith("l"))	Показывает, оканчиваются ли все элементы в списке thrill буквой "l" (возвращает true)
thrill.foreach(s => print(s))	Выполняет инструкцию print в отношении каждой строки в списке thrill (выводит "Willfilluntil")
thrill.foreach(print)	Делает то же самое, что и предыдущий код, но с использованием более лаконичной формы записи (также выводит "Willfilluntil")
thrill.head	Возвращает первый элемент в списке thrill (возвращает "Will")
Что используется	**Что этот метод делает**
thrill.init	Возвращает список всех элементов списка thrill, кроме последнего (возвращает List("Will", "fill"))
thrill.isEmpty	Показывает, не пуст ли список thrill (возвращает false)
thrill.last	Возвращает последний элемент в списке thrill (возвращает "until")
thrill.length	Возвращает количество элементов в списке thrill (возвращает 3)
thrill.map(s => s + "y")	Возвращает список, который получается в результате добавления "y" к каждому строковому элементу в списке thrill (возвращает List("Willy", "filly", "until y"))
thrill.mkString(", ")	Создает строку с элементами списка (возвращает "Will, fill, until")
thrill.filterNot(s => s.length == 4)	Возвращает список всех элементов в порядке их следования в списке thrill, за исключением имеющих длину 4 (возвращает List("until"))
thrill.reverse	Возвращает список, содержащий все элементы списка thrill, следующие в обратном порядке (возвращает List("until", "fill", "Will"))
thrill.sort((s, t) => s.charAt(0).toLower < t.charAt(0).toLower)	Возвращает список, содержащий все элементы списка thrill в алфавитном порядке с первым символом, преобразованным в символ нижнего регистра (возвращает List("fill", "until", "will"))
thrill.tail	Возвращает список thrill за минусом его первого элемента (возвращает List("fill", "until"))

Шаг 9. Применение кортежей

Еще одним полезным объектом-контейнером является *кортеж*. Кортежи, как и списки, нельзя изменять, но, в отличие от списков, в кортежах могут содержаться различные типы элементов. Список может быть типа `List[Int]` или `List[String]`, а кортеж

способен содержать одновременно как целые числа, так и строки. Кортежи находят широкое применение, например при возвращении из метода сразу нескольких объектов. Там, где на Java для хранения нескольких возвращаемых значений зачастую приходится создавать JavaBean-подобный класс, в Scala можно просто вернуть кортеж. Все делается просто: для создания экземпляра нового кортежа, содержащего объекты, нужно лишь заключить объекты в круглые скобки, отделив их друг от друга запятыми. После создания экземпляра кортежа доступ к его элементам можно получить, используя точку, знак подчеркивания и индекс элемента, причем подсчет элементов начинается с единицы. Пример показан в листинге 3.4.

Листинг 3.4. Создание и использование кортежа

```
val pair = (99, "Luftballons")
println(pair._1)
println(pair._2)
```

В первой строке листинга 3.4 создается новый кортеж, содержащий в качестве первого элемента целочисленное значение 99, а в качестве второго — строку `"Luftballons"`. Scala выводит тип кортежа в виде `Tuple2[Int, String]`, а также присваивает этот тип паре переменных. Во второй строке выполняется доступ к полю `_1`, в результате чего получается первый элемент 99. Символ точки (`.`) во второй строке аналогичен той точке, которая используется для доступа к полю или вызова метода. В данном случае выполняется доступ к полю по имени `_1`. Если запустим этот сценарий на выполнение, получим следующий результат:

```
99
Luftballons
```

Реальный тип кортежа зависит от количества содержащихся в

нем элементов и от типов этих элементов. Следовательно, типом кортежа `(99, "Luftballons")` является `Tuple2[Int, String]`, а типом кортежа `('u', 'r', "the", 1, 4, "me")` — `Tuple6[Char, Char, String, Int, Int, String]`[33].

> **Обращение к элементам кортежа**
>
> Возникает вопрос: а почему к элементам кортежа нельзя обратиться точно так же, как к элементам списка, например `pair(0)`? Дело в том, что используемый в списках метод `apply` всегда возвращает один и тот же тип, а в кортеже все элементы могут быть разных типов: у `_1` может быть один получаемый тип, у `_2` — другой и т. д. Эти числа вида `_N` начинаются с единицы, а не с нуля, поскольку начало отсчета с единицы традиционно используется в языках со статически типизированными кортежами, например `Haskell` и `ML`.

Шаг 10. Использование наборов и отображений

Поскольку Scala призван помочь вам получить преимущества как функционального, так и объектно-ориентированного стиля, в библиотеках его коллекций особое внимание обращают на разницу между изменяемыми и неизменяемыми коллекциями. Например, массивы всегда изменяемы, а списки неизменяемы. Scala также предоставляет изменяемые и неизменяемые альтернативы для наборов и отображений, но использует для обеих версий одни и те же простые имена. Для наборов и отображений Scala моделирует изменяемость в иерархии классов.

Например, в API Scala содержится основной *трейт* для наборов, который аналогичен Java-интерфейсу. (Более подробно трейты рассматриваются в главе 12.) Затем Scala предоставляет два

подчиненных трейта — один для изменяемых, а второй для неизменяемых наборов.

На рис. 3.2 показано, что для всех трех трейтов используется одно и то же простое имя `Set`. Но их полные имена отличаются друг от друга, поскольку все трейты размещаются в разных пакетах. Конкретный набор классов в Scala API, такой как набор, содержащий классы `HashSet` (см. рис. 3.2), является расширением либо изменяемого, либо неизменяемого трейта `Set`. (В то время как в Java вы реализуете интерфейсы, в Scala вы расширяете, или подмешиваете, трейты.) Следовательно, если нужно воспользоваться `HashSet`, можно в зависимости от потребностей выбирать между его изменяемой и неизменяемой разновидностями. Исходный способ создания набора показан в листинге 3.5.

Листинг 3.5. Создание, инициализация и использование неизменяемого набора

```
var jetSet = Set("Boeing", "Airbus")
jetSet += "Lear"
println(jetSet.contains("Cessna"))
```

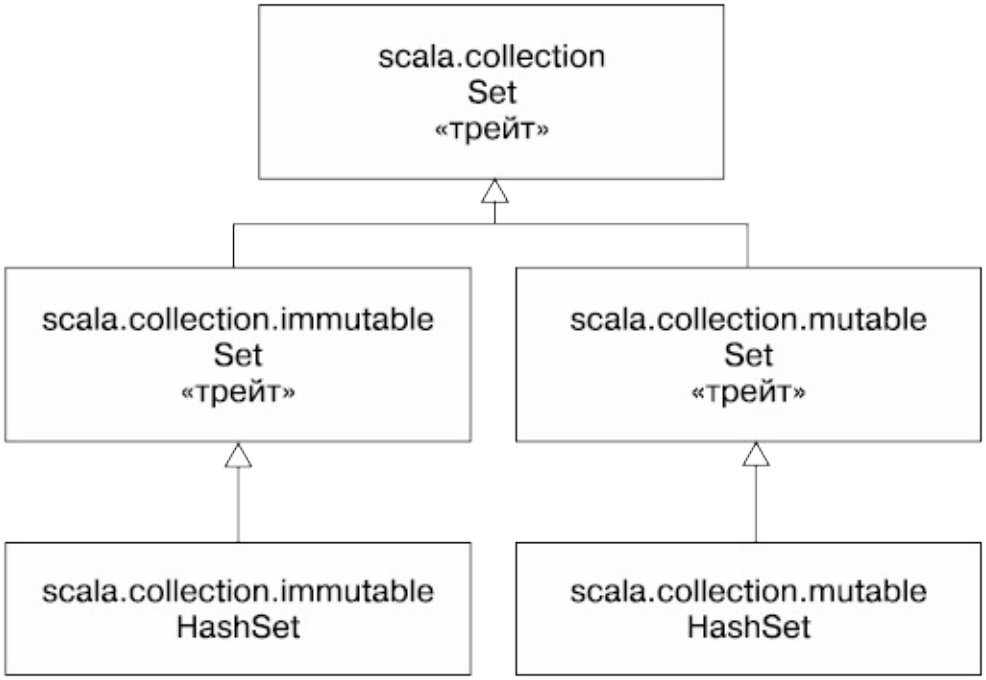


Рис. 3.2. Иерархия классов для наборов Scala

В первой строке кода листинга 3.5 определяется новая `var`-переменная по имени `jetSet`, которая инициализируется неизменяемым набором, содержащим две строки — `"Boeing"` и `"Airbus"`. Как показано в примере, в Scala наборы можно создавать точно так же, как списки и массивы, — путем вызова фабричного метода по имени `apply` в отношении объекта-спутника `Set`. В листинге 3.5 метод `apply` вызывается в отношении объекта-спутника для `scala.collection.immutable.Set`, возвращающего экземпляр исходного, неизменяемого класса `Set`. Компилятор Scala выводит тип переменной `jetSet`, определяя его как неизменяемый `Set[String]`.

Чтобы добавить к набору новый элемент, в отношении набора вызывается метод `+`, которому передается этот новый элемент. Как для изменяемых, так и для неизменяемых наборов метод `+` создает и возвращает новый набор с добавленным элементом. В листинге 3.5 работа ведется с неизменяемым набором. В изменяемых наборах предоставляется конкретный метод `+=`, однако для

неизменяемых наборов он не предлагается.

В данном случае вторая строка кода, `jetSet += "Lear`, фактически является сокращенной формой записи следующего кода:

```
jetSet = jetSet + "Lear"
```

Следовательно, во второй строке кода листинга 3.5 `var`-переменной `jetSet` присваивается новый набор, содержащий `"Boeing"`, `"Airbus"` и `"Lear"`. И наконец, в последней строке кода листинга 3.5 выводятся данные о том, содержится ли в наборе строка `"Cessna"`. (Как и ожидалось, выводится `false`.)

Если нужен изменяемый набор, следует, как показано в листинге 3.6, воспользоваться инструкцией `import`.

Листинг 3.6. Создание, инициализация и использование изменяемого набора

```
import scala.collection.mutable
val movieSet = mutable.Set("Hitch", "Poltergeist")
movieSet += "Shrek"
println(movieSet)
```

В первой строке листинга 3.6 импортируется изменяемый набор `Set`. Как и в Java, инструкция `import` позволяет использовать простое имя, например `Set`, вместо длинного полного имени. В результате при указании `Set` во второй строке компилятор знает, что подразумевается под `scala.collection.mutable.Set`. В этой строке `movieSet` инициализируется новым изменяемым набором, содержащим строки `"Hitch"` и `"Poltergeist"`. В следующей строке к изменяемому набору добавляется `"Shrek"`, для чего в отношении набора вызывается метод `+=` с передачей ему строки `"Shrek"`. Как уже упоминалось, `+=` является методом,

определенным в изменяемых наборах. Если хотите, можете вместо кода `movieSet += "Shrek"` воспользоваться кодом `movieSet.+= ("Shrek")`[34].

Хотя рассмотренной исходной реализации наборов, выполняемых изменяемыми и неизменяемыми фабричными методами `Set`, скорее всего, будет достаточно для большинства ситуаций, временами может потребоваться явный класс набора. К счастью, при этом используется аналогичный синтаксис. Нужно просто импортировать нужный класс и воспользоваться фабричным методом в отношении его объекта-спутника. Например, если нужен неизменяемый `HashSet`, можно сделать следующее:

```
import scala.collection.immutable.HashSet

val hashSet = HashSet("Tomatoes", "Chilies")
println(hashSet + "Coriander")
```

Еще одним полезным классом коллекций в Scala является отображение — `Map`. Как и для наборов, Scala предоставляет изменяемые и неизменяемые версии `Map` с использованием иерархии классов. Как показано на рис. 3.3, иерархия классов для отображений во многом похожа на иерархию для наборов. В пакете `scala.collection` есть основной трейт `Map` и два подчиненных трейта отображений `Map`: изменяемый вариант в `scala.collection.mutable` и неизменяемый вариант в `scala.collection.immutable`.

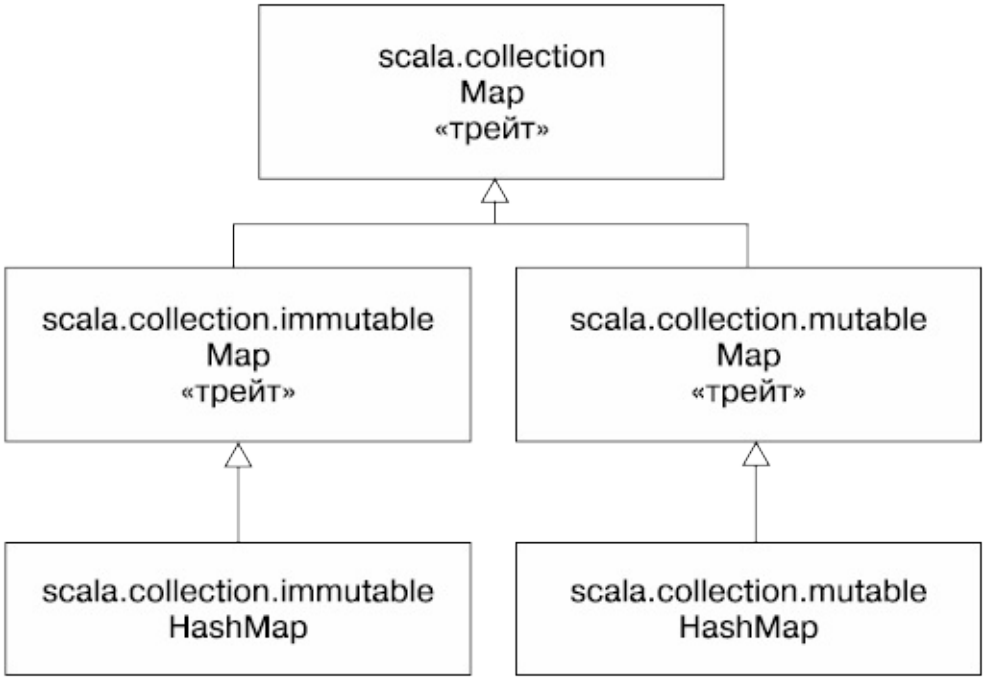


Рис. 3.3. Иерархия классов для Scala-отображений

Реализации `Map`, например `HashMap`-реализации в иерархии классов, показанной на рис. 3.3, расширяются либо в изменяемый, либо в неизменяемый трейт. Отображения можно создавать и инициализировать, используя фабричные методы, подобные тем, что применялись для массивов, списков и наборов.

Листинг 3.7. Создание, инициализация и использование изменяемого отображения

```
import scala.collection.mutable
val treasureMap = mutable.Map[Int, String]()
treasureMap += (1 -> "Go to island.")
treasureMap += (2 -> "Find big X on ground.")
treasureMap += (3 -> "Dig.")
println(treasureMap(2))
```

Например, в листинге 3.7 показана работа с изменяемым отображением. В первой строке импортируется изменяемое

отображение. Затем определяется val-переменная treasureMap, которая инициализируется пустым изменяемым отображением, имеющим целочисленные ключи и строковые значения. Отображение задается пустым, поскольку фабричному методу ничего не передается (в круглых скобках в Map[Int, String]() ничего не указано)[35]. В следующих трех строках к отображению добавляются пары «ключ — значение», для чего используются методы -> и +=. Как уже было показано, компилятор Scala преобразует выражения бинарных операций вида 1 -> "Go to island." в код (1).->("Go to island."). Следовательно, когда указывается 1 -> "Go to island.", фактически в отношении объекта 1 вызывается метод по имени ->, которому передается строка со значением "Go to island.". Метод ->, который можно вызвать в отношении любого объекта в программе Scala, возвращает двухэлементный кортеж, содержащий ключ и значение[36]. Затем этот кортеж передается методу += объекта отображения, на который ссылается treasureMap. И наконец, в последней строке выводится значение, соответствующее в treasureMap ключу 2.

При запуске этот код выведет следующие данные:

```
Find big X on ground.
```

Если отдать предпочтение неизменяемому отображению, то ничего импортировать не нужно, поскольку это отображение используется по умолчанию. Пример показан в листинге 3.8.

Листинг 3.8. Создание, инициализация и использование неизменяемого отображения

```
val romanNumeral = Map(
1 -> "I", 2 -> "II", 3 -> "III", 4 -> "IV", 5 ->
"V"
```

```
)
println(romanNumeral(4))
```

Учитывая отсутствие импортирования, при указании `Map` в первой строке листинга 3.8 вы получаете используемый по умолчанию экземпляр класса `scala.collection.immutable.Map`. Фабричному методу отображения передаются пять кортежей «ключ — значение», а он возвращает неизменяемое `Map`-отображение, содержащее переданные пары «ключ — значение». Если запустить код, показанный в листинге 3.8, он выведет `IV`.

Шаг 11. Обучение распознаванию функционального стиля

Как упоминалось в главе 1, Scala позволяет программировать в императивном стиле, но подталкивает вас перейти преимущественно к функциональному стилю. Если к Scala вы пришли с опытом работы в императивном стиле, к примеру, если приходилось программировать на Java, то одной из основных сложностей, с которой можно столкнуться, станет программирование в функциональном стиле. Мы понимаем, что поначалу этот стиль может быть неизвестен, и в данной книге стараемся перевести вас из одного состояния в другое. От вас также потребуются усилия, которые мы настоятельно рекомендуем приложить. Мы уверены, что при наличии опыта работы в императивном стиле изучение программирования в функциональном стиле не только позволит вам стать более квалифицированным программистом на Scala, но и расширит кругозор, сделав вас более ценным программистом в общем смысле.

Сначала нужно усвоить разницу между двумя стилями, отражающуюся в коде. Один верный признак заключается в том, что если код содержит `var`-переменные, то он, вероятнее всего, написан в императивном стиле. Если в коде вообще не содержатся

var-переменные, то есть в нем только val-переменные, то он, вероятнее всего, написан в функциональном стиле. Следовательно, одним из способов приближения к функциональному стилю является попытка обойтись в программах без var-переменных.

Обладая багажом императивности, то есть опытом работы с такими языками, как Java, C++ или C#, var-переменные можно рассматривать в качестве обычных переменных, а val-переменные — в качестве переменных особого вида. В то же время, если у вас имеется опыт работы в функциональном стиле на таких языках, как Haskell, OCaml или Erlang, val-переменные можно представлять как обычные, а var-переменные — как некое кощунственное обращение с кодом. Но с точки зрения Scala val- и var-переменные — всего лишь два разных инструмента в вашем арсенале средств и оба они одинаково полезны. Scala склоняет вас к применению val-переменных, но, по сути, дает возможность воспользоваться тем инструментом, который лучше подходит для решаемой задачи. И, тем не менее, даже если вы согласны с подобной философией, то поначалу можете испытывать трудности, связанные с избавлением от var-переменных в коде.

Рассмотрим позаимствованный из главы 2 пример цикла while, в котором используется var-переменная, означающая, что он выполнен в императивном стиле:

```
def printArgs(args: Array[String]): Unit = {
  var i = 0
  while (i < args.length) {
    println(args(i))
      i += 1
  }
}
```

Вы можете преобразовать этот код — придать ему более функциональный стиль, отказавшись от использования var-

переменной, например следующим образом:

```
def printArgs(args: Array[String]): Unit = {
    for (arg <- args)
        println(arg)
}
```

или вот так:

```
def printArgs(args: Array[String]): Unit = {
    args.foreach(println)
}
```

В этом примере демонстрируется одно из преимуществ программирования с меньшим количеством `var`-переменных. Реорганизованный (более функциональный) код выглядит понятнее, он более лаконичен, и в нем труднее допустить какие-либо ошибки, чем в исходном (более императивном) коде. Причина навязывания в Scala функционального стиля заключается в том, что он помогает создавать более понятный код, при написании которого сложнее ошибиться.

Но вы можете пойти еще дальше. Реорганизованный метод `printArgs` нельзя отнести к чисто функциональным, поскольку у него имеются побочные эффекты. В данном случае таким эффектом является вывод в поток стандартного устройства вывода. Признаком функции, имеющей побочные эффекты, является то, что типом результата у нее выступает `Unit`. Если функция не возвращает никакого интересного значения, о чем, собственно, и свидетельствует тип результата `Unit`, то единственным способом внесения этой функцией какого-либо изменения в окружающий мир является проявление какого-то побочного эффекта. Более функциональным подходом будет определение метода, форматирующего передаваемые аргументы с целью их последующего вывода на стандартное устройство и, как показано в листинге 3.9, просто возвращающего отформатированную строку.

Листинг 3.9. Функция без побочных эффектов или var-переменных

```
def formatArgs(args: Array[String]) =
args.mkString("\n")
```

Теперь вы действительно перешли на функциональный стиль: нет ни побочных эффектов, ни `var`-переменных. Метод `mkString`, который можно вызвать в отношении любой коллекции, допускающей последовательный перебор элементов (включая массивы, списки, наборы и отображения), возвращает строку, состоящую из результата вызова метода `toString` в отношении каждого элемента, с разделителями из переданной строки. Таким образом, если `args` содержит три элемента, `"zero"`, `"one"` и `"two"`, метод `formatArgs` возвращает `"zero\none\ntwo"`. Разумеется, эта функция, в отличие от методов `printArgs`, ничего не выводит, но для выполнения данной работы ее результаты можно легко передать функции `println`:

```
println(formatArgs(args))
```

У каждой полезной программы, вероятнее всего, будут какие-либо побочные эффекты. Отдавая предпочтение методам без побочных эффектов, вы будете стремиться к разработке программ, в которых такие эффекты сведены к минимуму. Одним из преимуществ подобного подхода станет упрощение тестирования ваших программ.

Например, чтобы протестировать любой из трех показанных ранее в этом разделе методов `printArgs`, вам придется переопределить метод `println`, перехватить передаваемый ему вывод и убедиться в том, что он соответствует вашим ожиданиям. В отличие от этого, функцию `formatArgs` можно протестировать, просто проверяя ее результат:

```
val res = formatArgs(Array("zero", "one", "two"))
assert(res == "zero\none\ntwo")
```

Имеющийся в Scala метод `assert` проверяет переданное ему булево выражение и, если оно вычисляется в `false`, выдает ошибку `AssertionError`. Если переданное булево выражение вычисляется в `true`, метод просто возвращает управление вызвавшему его коду. Более подробно о тестах, проводимых с помощью `assert`, и тестировании речь пойдет в главе 14.

И все-таки нужно иметь в виду, что ни `var`-переменные, ни побочные эффекты не следует рассматривать как нечто абсолютно неприемлемое. Scala не является чисто функциональным языком, заставляющим вас программировать все в функциональном стиле. Scala представляет собой гибрид императивного и функционального языка. Может оказаться, что в некоторых ситуациях для решения текущей задачи больше подойдет императивный стиль, и тогда вы должны им воспользоваться без всяких колебаний. Но чтобы помочь вам разобраться в программировании без применения `var`-переменных, в главе 7 будет показано множество конкретных примеров кода с использованием `var`-переменных и рассмотрены способы их преобразования в `val`-переменные.

Сбалансированная позиция для программистов, работающих на Scala

Отдавайте предпочтение `val`-переменным, неизменяемым объектам и методам без побочных эффектов. Старайтесь применять их в первую очередь. Используйте `var`-переменные, изменяемые объекты и методы с побочными эффектами при необходимости и наличии четкой обоснованности их применения.

Шаг 12. Считывание строк из файла

Сценарии, выполняющие небольшие повседневные задачи, часто нуждаются в обработке строк, взятых из файлов. В этом разделе будет создан сценарий, считывающий строки из файла и выводящий их на стандартное устройство, предваряя каждую строку количеством содержащихся в ней символов. Первая версия сценария показана в листинге 3.10.

Листинг 3.10. Считывание строк из файла

```
import scala.io.Source

if (args.length > 0) {
                      for          (line          <-
Source.fromFile(args(0)).getLines())
    println(line.length + " " + line)
}
else
  Console.err.println("Please enter filename")
```

Сценарий начинается с импорта класса `Source` из пакета `scala.io`. Затем он проверяет, указан ли в командной строке хотя бы один аргумент. Если да, то первый аргумент рассматривается как имя открываемого и обрабатываемого файла. Выражение `Source.fromFile(args(0))` пробует открыть указанный файл и возвращает объект типа `Source`, в отношении которого вызывается метод `getLines`. Этот метод возвращает значение типа `Iterator[String]`, в котором при каждой итерации предоставляется по одной строке с исключением символа конца строки.

Выражение `for` выполняет последовательный перебор этих строк и выводит длину каждой строки, затем пробел, а затем саму строку. Если аргументы в командной строке не указаны, финальное условие `else` выведет сообщение в поток стандартного устройства.

При добавлении этого кода в файл **countchars1.scala** и запуске его с указанием самого этого файла:

```
$ scala countchars1.scala countchars1.scala
```

вы увидите следующий текст:

```
22 import scala.io.Source
0
22 if (args.length > 0) {
0
51                              for      (line      <-
Source.fromFile(args(0)).getLines())
37         println(line.length + " " + line)
1 }
4 else
46              Console.err.println("Please   enter
filename")
```

Хотя сценарий в его текущем виде выводит необходимую информацию, может быть, вам захочется выстроить числа, выровняв их по правому краю, и добавить символ вертикальной черты, чтобы вид выводимой информации стал таким:

```
22 | import scala.io.Source
0 |
22 | if (args.length > 0) {
0 |
51      |                   for      (line      <-
Source.fromFile(args(0)).getLines())
37 |     println(line.length + " " + line)
1 | }
4 | else
46   |         Console.err.println("Please   enter
filename")
```

Чтобы реализовать задуманное, можно дважды выполнить последовательный перебор строк. При первом переборе определить максимальную ширину, требуемую какому-либо количеству символов в строке. А при втором переборе вывести данные, используя вычисленную ранее максимальную ширину. Поскольку перебор строк будет выполняться дважды, вы можете также присвоить эти строки переменной:

```
val lines = Source.fromFile(args(0)).getLines().toList
```

Завершающий выражение вызов метода `toList` нужен потому, что метод `getLines` возвращает итератор. По мере проведения итерации через итератор он истощается. Преобразование в список посредством вызова `toList` дает возможность выполнять итерацию любое количество раз без многократного выделения памяти для хранения всех строк из файла. Получается, что переменная `lines` ссылается на список строк с содержимым файла, указанного в командной строке. Затем, поскольку вам нужно вычислять ширину позиции для количества символов в каждой строке дважды — по одному разу за каждую итерацию, из этого выражения можно вывести небольшую функцию, подсчитывающую ширину, необходимую для символов, отображающих длину переданной строки:

```
def widthOfLength(s: String) = s.length.toString.length
```

Применяя эту функцию, максимальную ширину можно вычислить следующим образом:

```
var maxWidth = 0
for (line <- lines)
  maxWidth = maxWidth.max(widthOfLength(line))
```

Здесь с помощью выражения `for` выполняется

последовательный перебор всех строк, вычисляется ширина символов, показывающих длину строки, и, если она больше текущего максимума, ее значение присваивается var-переменной maxWidth, для которой было установлено начальное значение 0. (Метод max, который можно вызвать в отношении любого объекта типа Int, возвращает самое большое число, сравнивая значение, в отношении которого он был вызван, и значение, которое ему было передано.) В качестве альтернативного варианта, если предпочтительнее искать максимум без использования var-переменных, можно сначала определить самую длинную строку:

```
val longestLine = lines.reduceLeft(
  (a, b) => if (a.length > b.length) a else b
)
```

Метод reduceLeft применяет переданную ему функцию к первым двум элементам в списке lines, затем — к результату первого применения и к следующему элементу в lines и так далее до конца списка. Результатом каждого применения будет самая длинная строка из встреченных до сих пор, поскольку переданная функция (a, b) => if (a.length > b.length) a else b возвращает самую длинную из двух переданных строк. Метод reduceLeft вернет результат последнего применения функции, который в данном случае станет самым длинным строковым элементом списка lines.

Получив результат, можно вычислить максимальную ширину, передав самую длинную строку функции widthOfLength:

```
val maxWidth = widthOfLength(longestLine)
```

Теперь останется только вывести строки с надлежащим форматированием. Это можно сделать следующим образом:

```
for (line <- lines) {
  val numSpaces = maxWidth - widthOfLength(line)
```

```
    val padding = " " * numSpaces
    println(padding + line.length + " | " + line)
  }
```

В этом выражении `for` еще раз выполняет последовательный перебор элементов списка `lines`. Для каждой строки сначала вычисляется количество пробелов, устанавливаемых перед указанием длины строки, и это количество присваивается переменной `numSpaces` с помощью выражения `" " * numSpaces`. И наконец, выводится информация с надлежащим форматированием. Весь сценарий показан в листинге 3.11.

Листинг 3.11. Вывод отформатированного количества символов каждой строки файла

```
import scala.io.Source

def widthOfLength(s: String) = s.length.toString.length

if (args.length > 0) {

  val lines = Source.fromFile(args(0)).getLines().toList

  val longestLine = lines.reduceLeft(
      (a, b) => if (a.length > b.length) a else b
  )
  val maxWidth = widthOfLength(longestLine)

  for (line <- lines) {
    val numSpaces = maxWidth - widthOfLength(line)
    val padding = " " * numSpaces
```

```
      println(padding + line.length + " | " + line)
    }
  }
  else
    Console.err.println("Please enter filename")
```

Резюме

Знания, полученные в этой главе, позволят вам приступить к применению Scala для решения небольших задач, в особенности тех, для которых используются сценарии. В последующих главах рассмотренные темы мы изучим глубже, также будут представлены другие, не затронутые здесь темы.

29 Этот метод to фактически возвращает не массив, а иную последовательность, содержащую значения 0, 1 и 2, перебор которых выполняется выражением for. Последовательности и другие коллекции будут рассматриваться в главе 17.

30 Списки аргументов переменной длины или повторяющиеся параметры рассматриваются в разделе 8.8.

31 Применять запись new List не нужно, поскольку List.apply() определен в объекте-спутнике scala.List как фабричный метод. Более подробно объекты-спутники рассматриваются в разделе 4.3.

32 Причина, по которой в конце списка нужен Nil, заключается в том, что метод :: определен в классе List. Если попытаться просто воспользоваться кодом 1 :: 2 :: 3, то он не пройдет компиляцию, поскольку 3 относится к типу Int, у которого нет метода ::.

33 Хотя концептуально можно создавать кортежи любой длины, на данный момент библиотека Scala определяет их только до Tuple22.

34 Набор в листинге 3.6 изменяемый, поэтому повторно присваивать значение movieSet не нужно и данная переменная может относиться к val-переменным. В отличие от этого использование метода += с неизменяемым набором в листинге 3.5 требует повторного присваивания значения переменной jetSet, поэтому она должна быть var-переменной.

35 Явная параметризация типа требуется в листинге 3.7 из-за того, что без какого-либо значения, переданного фабричному методу, компилятор не в состоянии выполнить логический вывод типа параметров отображения. В отличие от этого, компилятор может выводить тип параметров из значений, переданных фабричному методу map, показанному в листинге 3.8, поэтому явного указания типа параметров там не требуется.

36 Используемый в Scala механизм, позволяющий вызывать метод -> в отношении любого объекта, — неявное преобразование — будет рассмотрен в главе 21.

4. Классы и объекты

В главах 2 и 3 вы разобрались в основах классов и объектов. В текущей главе предстоит углубленная проработка данной темы. Здесь будут приведены дополнительные сведения о классах, полях и методах, а также дано общее представление о том, что подразумевает использование точки с запятой. Также мы рассмотрим синглтон-объекты (объекты-одиночки) и способы их применения для написания и запуска приложений на Scala. Если вам уже знаком язык Java, вы увидите, что в Scala используются похожие, но все же немного отличающиеся концепции. Поэтому чтение данной главы пойдет на пользу даже большим знатокам языка Java.

4.1. Классы, поля и методы

Классы являются прообразами объектов. После определения класса из него как из прообраза можно создавать объекты, воспользовавшись для этого ключевым словом `new`. Например, при наличии следующего определения класса:

```
class ChecksumAccumulator {
    // сюда помещается определение класса
}
```

с помощью кода:

```
new ChecksumAccumulator
```

можно создавать объекты `ChecksumAccumulator`.

Внутри определения класса помещаются поля и методы, которые в общем называются *элементами* класса. Поля, определяемые либо как `val`-, либо как `var`-переменные, являются переменными, относящимися к объектам. Методы, которые

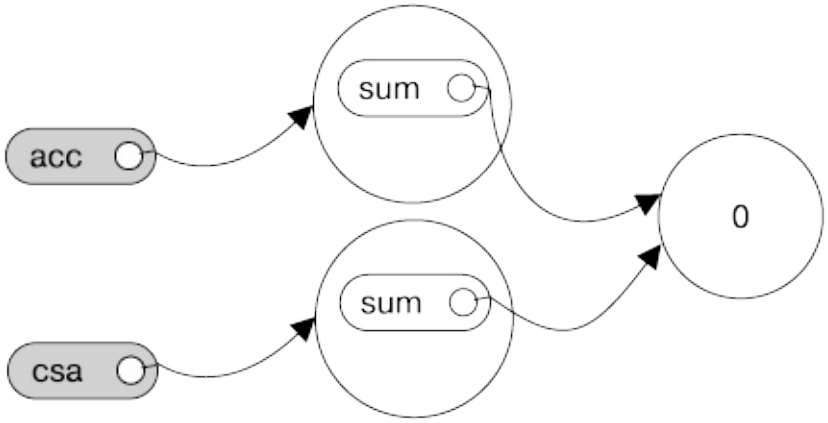
определяются с помощью ключевого слова `def`, содержат исполняемый код. В полях хранятся состояние или данные объекта, а методы используют эти данные для выполнения в отношении объекта вычислительной работы. При создании экземпляра класса система выполнения приложения резервирует часть памяти для хранения образа состояния получающегося при этом объекта (то есть содержимого его переменных). Например, если вы определите класс `ChecksumAccumulator` и дадите ему `var`-поле по имени `sum`:

```
class ChecksumAccumulator {
    var sum = 0
}
```

а потом дважды создадите его экземпляры с помощью следующего кода:

```
val acc = new ChecksumAccumulator
val csa = new ChecksumAccumulator
```

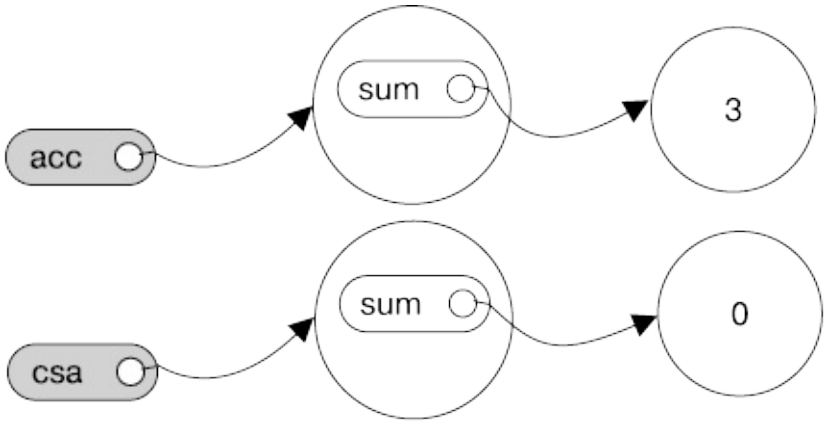
то образ объектов в памяти может получить такой вид:



Поскольку `sum` представляет собой поле, которое определено внутри класса `ChecksumAccumulator`, и относится к `var`-, а не к `val`-переменным, впоследствии полю `sum` может быть заново присвоено другое `Int`-значение:

```
acc.sum = 3
```

Теперь картинка может приобрести следующий вид:



По поводу рисунка нужно сделать следующее замечание: на нем показаны две переменные `sum`, одна из которых находится в объекте, на который ссылается `acc`, а другая — в объекте, на который ссылается `csa`. Поля также называют *переменными экземпляра*, поскольку каждый экземпляр получает собственный набор переменных. Все переменные экземпляра объекта составляют образ объекта в памяти. То, что здесь показано, свидетельствует не только о том, что есть две переменные `sum`, но и о том, что изменение одной из них никак не отражается на другой.

В этом примере следует также отметить, что у вас есть возможность изменить объект, на который ссылается `acc`, даже несмотря на то, что `acc` относится к `val`-переменным. Но с учетом того, что `acc` (или `csa`) являются `val`-, а не `var`-переменными, вы не можете присвоить им какой-нибудь другой объект. Например, следующая попытка не увенчается успехом:

```
// Не пройдет компиляцию, поскольку acc является
val-переменной
acc = new ChecksumAccumulator
```

Но зато вы можете рассчитывать на то, что переменная acc всегда будет ссылаться на тот же объект ChecksumAccumulat, с использованием которого вы ее инициализировали, но поля, содержащиеся внутри этого объекта, могут со временем измениться.

Одним из важных способов обеспечения надежности объекта является гарантия того, что состояние этого объекта (то есть значения его переменных экземпляра) остается корректным в течение всего его жизненного цикла. Первым шагом к предотвращению непосредственного стороннего доступа к полям является создание *закрытых* полей. Поскольку доступ к закрытым полям можно получить только методами, определенными в том же самом классе, весь код, который может обновить состояние, будет локализован в классе. Чтобы объявить поле закрытым, перед ним ставится модификатор доступа private:

```
class ChecksumAccumulator {
    private var sum = 0
}
```

С таким определением ChecksumAccumulator любая попытка доступа к sum за пределами класса будет неудачной:

```
val acc = new ChecksumAccumulator
acc.sum = 5 // Не пройдет компиляцию, поскольку
поле sum является закрытым
```

ПРИМЕЧАНИЕ

В Scala элементы класса делают открытыми, если нет явного указания какого-либо модификатора доступа. Иначе говоря, там, где в Java ставится модификатор public, в Scala вы обходитесь простым замалчиванием. Открытый (public) доступ в Scala является уровнем доступа по умолчанию.

Теперь, когда поле sum стало закрытым, доступ к нему может быть получен только из кода, определенного внутри тела самого класса. Следовательно, ChecksumAccumulator не будет широко использоваться ни одной из сторон, пока внутри этого класса мы не определим некоторые методы:

```
class ChecksumAccumulator {
    private var sum = 0

    def add(b: Byte): Unit = {
        sum += b
    }
    def checksum(): Int = {
        return ~(sum & 0xFF) + 1
    }
}
```

Теперь у ChecksumAccumulator есть два метода, add и checksum, и оба они демонстрируют основную форму определения функции, показанную на рис. 2.1.

Внутри этого метода могут использоваться любые параметры метода. Одной из важных характеристик параметров метода в Scala является то, что они относятся к val-, а не к var-переменным[37]. При попытке повторного присваивания значения параметру внутри метода в Scala произойдет сбой компиляции:

```
def add(b: Byte): Unit = {
     b = 1 // Не пройдет компиляцию, поскольку b
относится к val-переменным
    sum += b
}
```

Хотя методы add и checksum в данной версии ChecksumAccumulator реализуют желаемые функциональные

свойства вполне корректно, их можно определить в более лаконичном стиле. Во-первых, в конце метода `checksum` можно избавиться от лишнего слова `return`. В отсутствие явно указанной инструкции `return` метод в Scala возвращает последнее вычисленное им значение.

При написании методов рекомендуется применять стиль, исключающий явное и особенно многократное использование инструкций `return`. Каждый метод нужно рассматривать в качестве выражения, выдающего одно значение, которое и является возвращаемым. Эта философия будет мотивировать вас на создание небольших методов и разбиение слишком крупных методов на несколько мелких. В то же время выбор конструктивного решения зависит от контекста решаемых задач, и, если того требует обстановка, Scala упрощает написание методов, имеющих несколько явно указанных возвращаемых значений.

Поскольку метод `checksum` всего лишь вычисляет значение, ему не нужна явно указанная инструкция `return`. Еще одним сокращением для методов является отказ от использования фигурных скобок, если в методе вычисляется только одно выражение, дающее результат. Если результирующее выражение достаточно краткое, его можно даже поместить на той же строке, в которой указано ключевое слово `def`. Для максимальной лаконичности допускается не указывать тип возвращаемого результата, и тогда Scala выведет его самостоятельно. После внесения всех изменений класс `ChecksumAccumulator` приобретет следующий вид:

```
class ChecksumAccumulator {
  private var sum = 0
  def add(b: Byte) = sum += b
  def checksum() = ~(sum & 0xFF) + 1
}
```

Несмотря на то что компилятор Scala вполне корректно

выполнит вывод типов результатов методов add и checksum, показанных в предыдущем примере, читатели кода вынуждены будут вывести типы результатов путем *логических умозаключений* на основе изучения тел методов. Поэтому лучше все-таки будет всегда указывать типы результатов для открытых методов, объявленных в классе, в явном виде, даже когда компилятор может вывести их для вас самостоятельно. Применение этого стиля показано в листинге 4.1.

Листинг 4.1. Окончательная версия класса ChecksumAccumulator

```
// Этот код находится в файле ChecksumAccumulator.scala
class ChecksumAccumulator {
    private var sum = 0
    def add(b: Byte): Unit = { sum += b }
    def checksum(): Int = ~(sum & 0xFF) + 1
}
```

Методы с типом результата Unit, к которым относится и метод add класса ChecksumAccumulator, выполняются с целью получения побочного эффекта. Этот побочный эффект обычно определяется в виде изменения внешнего по отношению к методу состояния или в виде выполнения какой-либо операции ввода-вывода. Что касается метода add, то побочный эффект заключается в присваивании sum нового значения. Метод, который выполняется только для получения его побочного эффекта, называется *процедурой*.

4.2. Подразумеваемость использования точки с запятой

В программе на языке Scala точку с запятой в конце инструкции обычно можно не ставить. Если вся инструкция помещается на

одной строке, то, если хотите, можете поставить в конце этой строки точку с запятой, но делать это необязательно. В то же время точка с запятой нужна, если на одной строке размещаются сразу несколько инструкций:

```
val s = "hello"; println(s)
```

Если нужно набрать инструкцию, занимающую несколько строк, то в большинстве случаев вы можете просто ее ввести, а Scala разделит инструкции в нужном месте. Например, следующий код рассматривается как одна инструкция, расположенная на четырех строках:

```
if (x < 2)
    println("too small")
else
    println("ok")
```

И все же временами Scala разбивает инструкции на две части вопреки вашим желаниям:

```
x
+ y
```

Этот код рассматривается как две инструкции, `x` и `+y`. Если подразумевается, что он должен читаться как одна инструкция `x + y`, его нужно заключать в круглые скобки:

```
(x
+ y)
```

В качестве альтернативного варианта можно поместить `+` в конце строки. Именно поэтому при составлении цепочки из инфиксных операций, таких как `+`, в Scala обычно применяется стиль, предусматривающий помещение операторов в конец, а не в начало строки:

```
x +
y +
z
```

Правило, при соблюдении которого подразумевается постановка точки с запятой

Принцип работы точных правил разделения инструкций удивительно прост. Если вкратце, то конец строки рассматривается как точка с запятой, пока не возникнет одно из следующих условий.

1. Рассматриваемая строка заканчивается элементом, который не может законно находиться в конце строки, например точкой или инфиксным оператором.

2. Следующая строка начинается с элемента, с которого не может начинаться инструкция.

3. Строка заканчивается внутри круглых (...) или квадратных [...] скобок, поскольку там в любом случае не могут содержаться сразу несколько инструкций.

4.3. Синглтон-объекты

Как упоминалось в главе 1, один из аспектов, позволяющих Scala быть более объектно-ориентированным языком, чем Java, заключается в том, что в классах Scala не могут содержаться статические элементы. Вместо этого в Scala имеются *синглтон-объекты*. Определение синглтон-объекта выглядит так же, как определение класса, за исключением того, что вместо ключевого

слова `class` используется ключевое слово `object`. Пример показан в листинге 4.2.

Синглтон-объект в этом листинге называется `ChecksumAccumulator`, то есть у него такое же имя, как и у класса в предыдущем примере. Когда синглтон-объект использует общее с классом имя, то для класса он называется *объектом-спутником* или объектом-компаньоном. И класс, и его объект-спутник нужно определять в одном и том же исходном файле.

Класс по отношению к синглтон-объекту называется *классом-спутником*. Класс и его объект-спутник могут обращаться к закрытым элементам друг друга.

Листинг 4.2. Объект-спутник для класса ChecksumAccumulator

```
// Этот код находится в файле ChecksumAccumulator.scala
import scala.collection.mutable

object ChecksumAccumulator {

  private val cache = mutable.Map.empty[String, Int]

  def calculate(s: String): Int =
    if (cache.contains(s))
      cache(s)
    else {
      val acc = new ChecksumAccumulator
      for (c <- s)
        acc.add(c.toByte)
      val cs = acc.checksum()
      cache += (s -> cs)
```

```
            cs
        }
    }
```

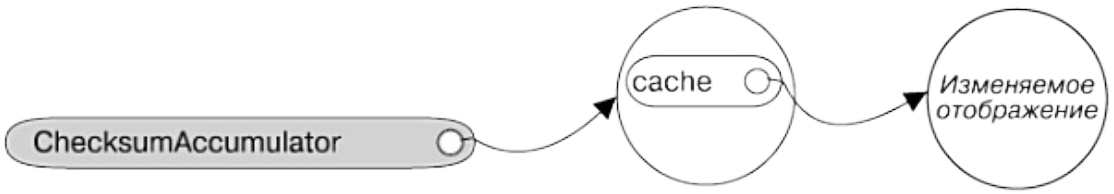
У синглтон-объекта `ChecksumAccumulator` имеется один метод по имени `calculate`, получающий значение типа `String` и вычисляющий контрольную сумму для символов типа `String`, находящихся в этом значении. У него также имеется одно закрытое поле `cache`, представленное изменяемым отображением, в котором кэшируются ранее вычисленные контрольные суммы[38]. В первой строке метода, `if (cache.contains(s))`, в `cache` определяется, не содержится ли переданная строка в отображении в качестве ключа. Если да, то просто возвращается отображенное на этот ключ значение `cache(s)`. В противном случае выполняется условие `else`, вычисляющее контрольную сумму. В первой строке условия `else` определяется `val`-переменная по имени `acc`, которая инициализируется новым экземпляром `ChecksumAccumulator`[39]. Во второй строке находится выражение `for`, выполняющее последовательный перебор каждого символа в переданной строке, преобразующее символ в значение типа `Byte` путем вызова в отношении этого символа метода `toByte` и передающее результат в метод `add` тех экземпляров `ChecksumAccumulator`, на которые ссылается `acc`. Когда завершится вычисление выражения `for`, в следующей строке метода в отношении `acc` будет вызван метод `checksum`, который берет контрольную сумму для переданного значения типа `String` и сохраняет ее в `val`-переменной по имени `cs`. В следующей строке, `cache += (s -> cs)`, переданный строковый ключ отображается на целочисленное значение контрольной суммы и эта пара «ключ — значение» добавляется к отображению `cache`. В последнем выражении метода, `cs`, обеспечивается использование контрольной суммы в качестве результата выполнения метода.

Если у вас есть опыт программирования на Java, то синглтон-

объекты можно представить в качестве хранилища для любых статических методов, которые можно было бы написать на Java. Методы в синглтон-объектах можно вызывать с использованием такого синтаксиса: имя синглтон-объекта, точка и имя метода. Например, метод `calculate` синглтон-объекта `ChecksumAccumulator` можно вызвать следующим образом:

```
ChecksumAccumulator.calculate("Every value is an
object")
```

Но синглтон-объект является не только хранилищем статических методов — это также объект первого класса. Поэтому имя синглтон-объекта можно рассматривать в качестве таблички с именем, прикрепленной к объекту.



Определение синглтон-объекта не является определением типа на том уровне абстракции, который используется в Scala. При наличии только определения объекта `ChecksumAccumulator` невозможно создать переменную типа `ChecksumAccumulator`. Точнее, тип с именем `ChecksumAccumulator` определяется классом-спутником синглтон-объекта. Тем не менее синглтон-объекты расширяют родительский класс и могут подмешивать трейты. Учитывая то, что каждый синглтон-объект является экземпляром своего родительского класса и подмешанных в него трейтов, его методы можно вызывать через эти типы, ссылаясь на него из переменных этих типов и передавая ему методы, ожидающие использования этих типов. Примеры синглтон-объектов, являющихся наследниками классов и трейтов, показаны в главе 13.

Одно из отличий классов от синглтон-объектов состоит в том,

что синглтон-объекты не могут принимать параметры, а классы — могут. Поскольку создать экземпляр синглтон-объекта с помощью ключевого слова `new` нельзя, передать ему параметры не представляется возможным. Каждый синглтон-объект реализуется как экземпляр *синтетического класса*, ссылка на который находится в статической переменной, поэтому у них и у статических классов Java одинаковая семантика инициализации[40]. В частности, синглтон-объект инициализируется при первом обращении к нему какого-либо кода.

Синглтон-объект, который не имеет общего имени с классом-спутником, называется *автономным объектом*. Такие объекты можно применять для решения многих задач, включая сбор в одно целое родственных вспомогательных методов или определение точки входа в приложение Scala. Именно этот случай и будет рассмотрен в следующем разделе.

4.4. Приложение на языке Scala

Для запуска программы на Scala нужно представить имя автономного синглтон-объекта с методом `main`, получающим один параметр с типом `Array[String]` и использующим тип результата `Unit`. Точкой входа в приложение может стать любой автономный объект с методом `main`, имеющим надлежащую сигнатуру. Пример показан в листинге 4.3:

Листинг 4.3. Приложение Summer

```
// Код находится в файле Summer.scala
import ChecksumAccumulator.calculate

object Summer {
  def main(args: Array[String]) = {
    for (arg <- args)
```

```
        println(arg + ": " + calculate(arg))
    }
  }
```

Синглтон-объект, показанный в листинге 4.3, называется `Summer`. Его метод `main` имеет надлежащую сигнатуру, поэтому его можно использовать в качестве приложения. Первая инструкция в файле импортирует метод `calculate`, который определен в объекте `ChecksumAccumulator` из предыдущего примера. Инструкция `import` позволяет далее использовать в файле простое имя метода[41]. Тело метода `main` всего лишь выводит на стандартное устройство каждый аргумент и контрольную сумму для аргумента, разделяя их двоеточием.

ПРИМЕЧАНИЕ

Подразумевается, что в каждый свой исходный файл Scala импортирует элементы пакетов java.lang и scala, а также элементы синглтон-объекта по имени Predef. В Predef, который находится в пакете scala, содержится множество полезных методов. Например, когда в исходном файле Scala встречается println, фактически вызывается println из Predef. (А метод Predef.println в свою очередь вызывает метод Console.println, который фактически и выполняет всю работу.) Когда же встречается assert, вызывается метод Predef.assert.

Для запуска приложения Summer поместите код из листинга 4.3 в файл по имени `Summer.scala`. Поскольку в Summer используется `ChecksumAccumulator`, добавьте код для `ChecksumAccumulator` как для класса, показанного в листинге 4.1, так и для его объекта-спутника, представленного в листинге 4.2, в файл `ChecksumAccumulator.scala`.

Одним из отличий Scala от Java является то, что в Java от вас

требуется поместить открытый класс в файл, названный по имени класса, например класс SpeedRacer нужно поместить в файл SpeedRacer.java, а в Scala файл с расширением .scala можно называть как угодно независимо от того, какие классы Scala или код в них помещаются. Но, как правило, когда речь идет не о сценариях, рекомендуется придерживаться стиля, при котором файлы называются по именам классов, которые в них содержатся, как это делается в Java, чтобы программистам было легче искать классы по именам их файлов. Именно этим подходом мы и воспользовались в отношении двух файлов в данном примере. Имеются в виду файлы Summer.scala и ChecksumAccumulator.scala.

Ни ChecksumAccumulator.scala, ни Summer.scala не являются сценариями, поскольку они заканчиваются в определении. В отличие от этого, сценарий должен заканчиваться в выражении, выдающем результат. Поэтому при попытке запуска Summer.scala в качестве сценария интерпретатор Scala пожалуется на то, что Summer.scala не заканчивается в выражении, выдающем результат (конечно, если предположить, что вы самостоятельно не добавили какое-либо выражение после определения объекта Summer). Вместо этого нужно будет откомпилировать эти файлы с помощью компилятора Scala, а затем запустить получившиеся в результате файлы классов. Для этого можно воспользоваться основным компилятором Scala по имени scalac:

```
$ scalac ChecksumAccumulator.scala Summer.scala
```

Эта команда откомпилирует ваши исходные файлы, но завершиться компиляция может после весьма ощутимой задержки. Дело в том, что при каждом запуске компилятора, до того как внимание привлекут переданные вами новоиспеченные исходные файлы, нужно потратить время на сканирование содержимого jar-файлов и выполнение другой предварительной работы. Поэтому в

дистрибутив Scala включается работающий в фоновом режиме Scala-компилятор под названием `fsc` (от fast Scala compiler — быстрый компилятор Scala). Он используется следующим образом:

```
$ fsc ChecksumAccumulator.scala Summer.scala
```

При первом запуске `fsc` будет создан локальный фоновый сервер (демон), прикрепленный к порту вашего компьютера. Затем через этот порт будут отправляться списки компилируемых файлов, а фоновая программа станет заниматься их компиляцией. При следующем запуске `fsc` фоновая программа уже будет запущена, поэтому `fsc` просто отправит ей список файлов, а она без промедления скомпилирует их. Используя `fsc`, дожидаться запуска среды выполнения Java придется лишь в первый раз. Если понадобится остановить фоновую программу `fsc`, можно будет воспользоваться командой `fsc -shutdown`.

Запуск любой из команд — `scalac` или `fsc` — приведет к созданию файлов классов Java, которые затем можно будет запускать через команду `scala` — ту же самую, которая использовалась вами для вызова интерпретатора в предыдущих примерах. Но вместо передачи интерпретатору имени файла с расширением `.scala`, который содержит предназначенный для интерпретации код Scala, как это делалось во всех предыдущих примерах[42], в данном случае ему нужно будет передать имя автономного объекта, содержащего метод `main`, имеющий надлежащую сигнатуру. Следовательно, приложение Summer может быть запущено путем набора команды:

```
$ scala Summer of love
```

Вы сможете увидеть контрольные суммы, выведенные для двух аргументов командной строки:

```
of: -213
love: -182
```

4.5. Трейт App

В Scala имеется трейт `scala.App`, призванный экономить время, которое вы тратите на набор текста. Хотя мы еще не рассматривали все, что нужно для того, чтобы можно было досконально разобраться в работе трейтов, мы все же решили, что об этом трейте вам захочется узнать прямо сейчас. Пример его использования показан в листинге 4.4.

Листинг 4.4. Использование трейта App

```
import ChecksumAccumulator.calculate

object FallWinterSpringSummer extends App {

  for (season <- List("fall", "winter", "spring"))
    println(season + ": " + calculate(season))
}
```

Чтобы воспользоваться трейтом, сначала нужно указать после имени вашего синглтон-объекта инструкцию расширения `extends App`. Затем вместо написания метода `main` между фигурными скобками синглтон-объекта следует набрать код, который вы поместили бы непосредственно в метод `main`. Обратиться к аргументам командной строки можно через массив строковых элементов по имени `args`. Вот и все. Это приложение можно откомпилировать и запустить точно так же, как и любое другое.

Резюме

В этой главе рассматривались основы классов и объектов в Scala и были показаны приемы компиляции и запуска приложений. Далее мы познакомимся с основными типами данных и порядком их использования.

37 Причина использования в качестве параметров val-переменных заключается в более легком составлении представления об этих переменных. Такую переменную больше не нужно отслеживать на предмет присваивания ей нового значения, как это было бы при использовании var-переменной.

38 Здесь cache используется, чтобы показать синглтон-объект с полем. Кэширование с помощью поля cache помогает оптимизировать быстродействие, сокращая за счет расхода памяти время вычисления и разменивая расход памяти на время вычисления. Как правило, использовать таким образом кэш-память целесообразно только в том случае, если с ее помощью можно решить проблемы быстродействия и воспользоваться отображением со слабыми ссылками, например WeakHashMap в scala.collection.jcl, чтобы записи в кэш-памяти могли попадать в сборщик мусора при дефиците памяти.

39 Поскольку ключевое слово new используется только для создания экземпляров классов, новый объект, созданный здесь в качестве экземпляра класса ChecksumAccumulator, не является одноименным синглтон-объектом.

40 В качестве имени синтетического класса используется имя объекта со знаком доллара. Следовательно, синтетический класс, применяемый для синглтон-объекта ChecksumAccumulator, называется ChecksumAccumulator$.

41 При наличии опыта программирования на Java такой импорт можно сопоставить с объявлением статического импорта, введенного в Java 5. Единственным отличием является то, что в Scala импортировать элементы можно из любого объекта, а не только из синглтон-объектов.

42 Механизм, который программа Scala использует для интерпретации исходного файла Scala, заключается в том, что она компилирует исходный код Scala в байт-коды Java, тут же загружает их с помощью загрузчика класса и приступает к их выполнению.

5. Основные типы и операции

После того как были рассмотрены в действии классы и объекты, самое время поглубже изучить имеющиеся в Scala основные типы и операции. Если вы хорошо знакомы с Java, то вас может обрадовать тот факт, что в Scala основные типы и операторы имеют такие же значения, что и в Java. И все же есть интересные различия, ради которых с этой главой стоит ознакомиться даже тем, кто считает себя опытным разработчиком Java-приложений. Поскольку некоторые аспекты Scala, рассматриваемые в этой главе, в основном такие же, как и в Java, мы указываем, какие разделы Java-разработчики могут пропустить.

В этой главе будет дан обзор основных типов Scala, включая строки типа `String` и типы значений `Int`, `Long`, `Short`, `Byte`, `Float`, `Double`, `Char` и `Boolean`. Мы также рассмотрим операции, которые могут выполняться в отношении этих типов, и вопросы соблюдения приоритета операторов в выражениях Scala. Поговорим мы и о том, как подразумеваемые преобразования могут обогатить варианты основных типов, позволяя выполнять дополнительные операции вдобавок к тем, что поддерживаются в Java.

5.1. Некоторые основные типы

В табл. 5.1 показан ряд основных типов, используемых в Scala, а также диапазоны значений, которые могут принимать экземпляры этих типов. В совокупности типы `Byte`, `Short`, `Int`, `Long` и `Char` называются *целочисленными типами*. Целочисленные типы плюс `Float` и `Double` именуются *числовыми типами*.

Таблица 5.1. Некоторые основные типы

Основной	

тип	Диапазон
Byte	8-разрядное целое число со знаком и с дополнением до двух (от -2^7 до $2^7 - 1$ включительно)
Short	16-разрядное целое число со знаком и с дополнением до двух (от -2^{15} до $2^{15} - 1$ включительно)
Int	32-разрядное целое число со знаком и с дополнением до двух (от -2^{31} до $2^{31} - 1$ включительно)
Основной тип	**Диапазон**
Long	64-разрядное целое число со знаком и с дополнением до двух (от -2^{63} до $2^{63} - 1$ включительно)
Char	16-разрядный беззнаковый Unicode-символ (от 0 до $2^{16} - 1$ включительно)
String	Последовательность Char-значений
Float	32-разрядное число с плавающей точкой одинарной точности, соответствующее стандарту IEEE 754
Double	64-разрядное число с плавающей точкой двойной точности, соответствующее стандарту IEEE 754
Boolean	true или false

За исключением типа `String`, который находится в пакете `java.lang`, все типы, показанные в табл. 5.1, входят в пакет `scala`[43]. Например, полное имя типа `Int` обозначается `scala.Int`. Но, учитывая, что все элементы пакета `scala` и `java.lang` автоматически импортируются в каждый исходный файл Scala, можно повсеместно использовать только простые имена, то есть имена вида `Boolean`, `Char` или `String`.

Опытные Java-разработчики заметят, что основные типы Scala имеют в точности такие же диапазоны, как и соответствующие им типы в Java. Это позволяет компилятору Scala в создаваемом им байт-коде преобразовывать экземпляры *типов значений* Scala, например `Int` или `Double`, в элементарные типы Java.

5.2. Литералы

Все основные типы, перечисленные в табл. 5.1, могут быть записаны с помощью *литералов*. Литерал представляет собой способ записи постоянного значения непосредственно в коде.

Ускоренный режим чтения для Java-программистов

Синтаксис большинства литералов, показанных в данном разделе, совпадает с синтаксисом, применяемым в Java, поэтому знатоки Java могут спокойно пропустить практически весь раздел. Некоторые различия, о которых стоит прочитать, касаются используемых в Scala неформатированных строк и символов (рассматриваются в подразделе «Строковые литералы»), а также строковой интерполяции. Кроме того, в Scala не поддерживаются восьмеричные литералы, а целочисленные литералы, начинающиеся с нуля, например `031`, не проходят компиляцию.

Целочисленные литералы

Целочисленные литералы для типов `Int`, `Long`, `Short` и `Byte` используются в двух видах — десятичном и шестнадцатеричном. Способ, применяемый для начала записи целочисленного литерала, показывает основание числа. Если число начинается с `0x` или `0X`, то оно шестнадцатеричное (по основанию 16) и может содержать цифры от 0 до 9, а также буквы от A до F в верхнем или нижнем регистре, которыми отображаются цифры. Примеры использования выглядят следующим образом:

```
scala> val hex = 0x5
hex: Int = 5

scala> val hex2 = 0x00FF
hex2: Int = 255
```

```
scala> val magic = 0xcafebabe
magic: Int = -889275714
```

Обратите внимание на то, что оболочка Scala всегда выводит целочисленные значения в числах по основанию 10 независимо от формы литерала, которой вы могли воспользоваться для инициализации этих значений. Таким образом, интерпретатор показывает значение переменной `hex2`, которая была инициализирована с использованием литерала `0x00FF`, как десятичное число `255`. (Разумеется, не нужно все принимать на веру. Хорошим способом приступить к освоению языка станет практическая работа с этими инструкциями в интерпретаторе по мере чтения данной главы.) Если цифра, с которой начинается число, не нуль и не имеет никаких других знаков отличия, значит число десятичное (по основанию 10), например:

```
scala> val dec1 = 31
dec1: Int = 31

scala> val dec2 = 255
dec2: Int = 255

scala> val dec3 = 20
dec3: Int = 20
```

Если целочисленный литерал заканчивается на `L` или `l`, то он показывает число типа `Long`, в противном случае это число относится к типу `Int`. Посмотрите на примеры целочисленных литералов `Long`:

```
scala> val prog = 0XCAFEBABEL
prog: Long = 3405691582

scala> val tower = 35L
```

```
tower: Long = 35

scala> val of = 31l
of: Long = 31
```

Когда Int-литерал присваивается переменной типа Short или Byte, он рассматривается как принадлежащий к типу Short или Byte, если, конечно, его значение находится внутри допустимого для данного типа диапазона, например:

```
scala> val little: Short = 367
little: Short = 367
scala> val littler: Byte = 38
littler: Byte = 38
```

Литералы чисел с плавающей точкой

Литералы чисел с плавающей точкой состоят из десятичных цифр, которые также могут содержать необязательный символ десятичной точки, и после них может стоять необязательный символ E или e и экспонента. Посмотрите на примеры литералов чисел с плавающей точкой:

```
scala> val big = 1.2345
big: Double = 1.2345

scala> val bigger = 1.2345e1
bigger: Double = 12.345

scala> val biggerStill = 123E45
biggerStill: Double = 1.23E47
```

Обратите внимание на то, что экспонента означает степень числа 10, на которую умножается остальная часть числа.

Следовательно, `1.2345e1` равняется числу 1,2345, умноженному на 10^1, откуда получается число 12,345. Если литерал числа с плавающей точкой заканчивается на `F` или `f`, значит число относится к типу `Float`, в противном случае оно принадлежит к типу `Double`. Дополнительно литералы чисел с плавающей точкой могут заканчиваться на `D` или `d`. Посмотрите на примеры литералов чисел с плавающей точкой:

```
scala> val little = 1.2345F
little: Float = 1.2345

scala> val littleBigger = 3e5f
littleBigger: Float = 300000.0
```

Последнее значение, выраженное как тип `Double`, может принимать также иную форму:

```
scala> val anotherDouble = 3e5
anotherDouble: Double = 300000.0

scala> val yetAnother = 3e5D
yetAnother: Double = 300000.0
```

Символьные литералы

Символьные литералы состоят из любого Unicode-символа, заключенного в одинарные кавычки:

```
scala> val a = 'A'
a: Char = A
```

Кроме того что символ представляется в одинарных кавычках в явном виде, его можно идентифицировать с использованием кода из таблицы символов Unicode. Для этого нужно записать `\u`, после

чего указать четыре шестнадцатеричные цифры кода:

```
scala> val d = '\u0041'
d: Char = A

scala> val f = '\u0044'
f: Char = D
```

Такие символы в кодировке Unicode могут появляться в любом месте программы на языке Scala. Например, вы можете набрать следующий идентификатор:

```
scala> val B\u0041\u0044 = 1
BAD: Int = 1
```

Он рассматривается точно так же, как идентификатор BAD, являющийся результатом раскрытия символов в кодировке Unicode в показанном ранее коде. По сути, в именовании идентификаторов таким образом нет ничего хорошего, поскольку их трудно прочесть. Иногда этот синтаксис может позволить исходным файлам Scala, содержащим отсутствующие в таблице ASCII символы из таблицы Unicode, быть представленными в кодировке ASCII.

И наконец, нужно упомянуть о нескольких символьных литералах, представленных специальными управляющими последовательностями (escape sequences), показанными в табл. 5.2, например:

```
scala> val backslash = '\\'
backslash: Char = \
```

Таблица 5.2. Управляющие последовательности специальных символьных литералов

Литерал	Предназначение

\n	Перевод строки (\u000A)
\b	Возврат на одну позицию (\u0008)
\t	Табуляция (\u0009)
\f	Перевод страницы (\u000C)
\r	Возврат каретки (\u000D)
\"	Двойная кавычка (\u0022)
\'	Одинарная кавычка (\u0027)
\\	Обратный слеш (\u005C)

Строковые литералы

Строковый литерал состоит из символов, заключенных в двойные кавычки:

```
scala> val hello = "hello"
hello: String = hello
```

Синтаксис символов внутри кавычек такой же, как и в символьных литералах, например:

```
scala> val escapes = "\\\"\'"
escapes: String = \"'
```

Поскольку этот синтаксис неудобен для строк, содержащих множество управляющих последовательностей, или для строк, не умещающихся в одну строку текста, для неформатированных строк в Scala включен специальный синтаксис. Неформатированная строка начинается и заканчивается тремя идущими подряд двойными кавычками (""). Внутри неформатированной строки могут содержаться любые символы, включая символы новой строки, кавычки и специальные символы, за исключением, разумеется, трех кавычек подряд. Например, следующая программа выводит сообщение, используя неформатированную строку:

```
println("""Welcome to Ultamix 3000.
           Type "HELP" for help.""")
```

Но при запуске этого кода получается не совсем то, что хотелось:

```
Welcome to Ultamix 3000.
           Type "HELP" for help.
```

Проблема во включении в строку пробелов перед второй строкой текста! Чтобы справиться с этой весьма часто возникающей ситуацией, вы можете вызвать в отношении строк метод `stripMargin`. Для этого поставьте символ вертикальной черты (`|`) перед каждой строкой текста, а затем в отношении всей строки вызовите метод `stripMargin`:

```
println("""|Welcome to Ultamix 3000.
           |Type "HELP" for help.""".stripMargin)
```

Вот теперь код ведет себя подобающим образом:

```
Welcome to Ultamix 3000.
Type "HELP" for help.
```

Литералы обозначений

Литерал обозначения записывается как 'ident, где ident может быть любым буквенно-цифровым идентификатором. Такие литералы отображаются на экземпляры предопределенного класса `scala.Symbol`. Например, литерал `'cymbal` будет расширен компилятором в вызов фабричного метода `Symbol("cymbal")`. Литералы обозначений обычно используются в ситуациях, при которых в динамически типизованных языках вы бы воспользовались просто идентификатором. Например, может потребоваться определить метод, обновляющий запись в базе данных:

```
scala> def updateRecordByName(r: Symbol, value:
Any) = {
         // сюда помещается код
       }
updateRecordByName: (Symbol,Any)Unit
```

Метод получает в качестве параметров обозначение, указывающее на имя поля, в которое ведется запись, и значение, которым это поле будет обновлено в записи. В динамически типизованных языках такую операцию можно вызвать, передавая методу необъявленный идентификатор поля. Но в Scala следующий код не пройдет компиляцию:

```
scala>    updateRecordByName(favoriteAlbum,    "OK
Computer")
<console>:6: error: not found: value favoriteAlbum
              updateRecordByName(favoriteAlbum,  "OK
Computer")
                                 ^
```

Вместо этого практически с такой же лаконичностью можно передать литерал обозначения:

```
scala>    updateRecordByName('favoriteAlbum,    "OK
Computer")
```

С обозначением, кроме как узнать его имя, сделать ничего нельзя:

```
scala> val s = 'aSymbol
s: Symbol = 'aSymbol

scala> val nm = s.name
nm: String = aSymbol
```

Следует также заметить, что обозначения характеризуются

изолированностью. Если набрать один и тот же литерал обозначения дважды, то оба выражения будут ссылаться на один и тот же `Symbol`-объект.

Булевы литералы

У типа `Boolean` имеются два литерала — `true` и `false`:

```
scala> val bool = true
bool: Boolean = true

scala> val fool = false
fool: Boolean = false
```

Вот, собственно, и все. Теперь вы буквально (или литерально) стали большим специалистом по Scala.

5.3. Строковая интерполяция

В Scala включен довольно гибкий механизм для строковой интерполяции, позволяющий вставлять выражения в строковые литералы. В самом распространенном случае использования этот механизм предоставляет лаконичную и удобочитаемую альтернативу объединению строк. Рассмотрим пример:

```
val name = "reader"
println(s"Hello, $name!")
```

Выражение `s"Hello, $name!"` является *обрабатываемым* строковым литералом. Поскольку за буквой `s` стоит открывающая кавычка, Scala для обработки литерала воспользуется *строковым интерполятором* `s`. Интерполятор `s` станет вычислять каждое встроенное выражение, вызывая в отношении всех результатов метод `toString` и заменяя встроенные выражения в литерале

этими результатами. Таким образом, из `s"Hello, $name!"` получится `"Hello, reader!"`, то есть точно такой же результат, как при использовании кода `"Hello, " + name + "!"`.

После знака доллара ($) в обрабатываемом строковом литерале можно указать любое выражение. Для выражений с одной переменной зачастую можно просто поместить после знака доллара имя этой переменной. Все символы, вплоть до первого, не относящегося к идентификатору, Scala будет интерпретировать как выражение. Если в выражение включены символы, не являющиеся идентификаторами, это выражение следует заключить в фигурные скобки, а открывающая фигурная скобка должна ставиться сразу же после знака доллара, например:

```
scala> s"The answer is ${6 * 7}."
res0: String = The answer is 42.
```

В Scala имеются еще два строковых интерполятора — `raw` и `f`. Строковый интерполятор `raw` ведет себя практически так же, как и `s`, за исключением того, что не распознает управляющие последовательности символьных литералов (те самые, которые показаны в табл. 5.2). Например, следующая инструкция выводит четыре, а не два обратных слеша:

```
println(raw"No\\\\escape!")      // выводит:
No\\\\escape!
```

Строковый интерполятор `f` позволяет прикреплять к встроенным выражениям инструкции форматирования в стиле функции `printf`. Инструкции ставятся после выражения и начинаются со знака процента (%), при этом используется синтаксис, заданный классом `java.util.Formatter`. Например, вот как можно было бы отформатировать число π:

```
scala> f"${math.Pi}%.5f"
res1: String = 3.14159
```

Если для встроенного выражения не указать никаких инструкций форматирования, строковый интерполятор `f` по умолчанию превратится в `%s`, что означает подстановку значения, полученного в результате выполнения метода `toString`, точно так же, как это делает строковый интерполятор `s`, например:

```
scala> val pi = "Pi"
pi: String = Pi
scala> f"$pi is approximately ${math.Pi}%.8f."
res2: String = Pi is approximately 3.14159265.
```

В Scala строковая интерполяция реализуется перезаписью кода в ходе компиляции. Компилятор в качестве выражения строкового интерполятора будет рассматривать любое выражение, состоящее из идентификатора, за которым сразу же стоит открывающая двойная кавычка строкового литерала. Строковые интерполяторы `s`, `f` и `raw` реализуются посредством этого общего механизма. Библиотеки и пользователи могут определять другие строковые интерполяторы, применяемые в иных целях.

5.4. Все операторы являются методами

Для основных типов в Scala имеется весьма богатый набор операторов. Как упоминалось в предыдущих главах, эти операторы являются привлекательным синтаксисом для обычных вызовов методов. Например, `1 + 2` означает то же самое, что и `1.+(2)`. Иными словами, в классе `Int` имеется метод по имени `+`, который получает `Int`-значение и возвращает `Int`-результат. Этот метод `+` вызывается при сложении двух `Int`-значений:

```
scala> val sum = 1 + 2 // Scala вызывает 1.+(2)
sum: Int = 3
```

Чтобы убедиться в этом, можете набрать выражение, в точности

соответствующее вызову метода:

```
scala> val sumMore = 1.+(2)
sumMore: Int = 3
```

Фактически в классе Int содержится несколько перегружаемых методов +, получающих различные типы параметров[44]. Например, у Int имеется еще один метод, также с именем +, который получает и возвращает значения типа Long. При сложении Long и Int будет вызван именно этот альтернативный метод:

```
scala> val longSum = 1 + 2L // Scala вызывает 1.+
(2L)
longSum: Long = 3
```

Символ + является оператором, а если точнее — инфиксным оператором. Система записи операторов не ограничивается такими методами, как +, которые в других языках выглядят как операторы. В нотации операторов можно воспользоваться любым методом. Например, в классе String есть метод indexOf, получающий один параметр типа Char. Метод indexOf ведет поиск первого появления в строке указанного символа и возвращает его индекс или -1, если символ найден не будет. Метод indexOf можно использовать как оператор:

```
scala> val s = "Hello, world!"
s: String = Hello, world!

scala> s indexOf 'o' // Scala вызывает
s.indexOf('o')
res0: Int = 4
```

Кроме того, в классе String предлагается перегружаемый метод indexOf, получающий два параметра: символ, поиск которого будет вестись, и индекс, с которого нужно начинать

поиск. (Другой метод, только что показанный `indexOf`, начинает поиск с нулевого индекса, то есть с начала значения типа `String`.) Несмотря на то что данный метод `indexOf` получает два аргумента, его также можно применять, используя систему записи операторов. Но когда при этом вызывается метод, получающий несколько аргументов, их нужно заключать в круглые скобки. Например, вот как эта другая форма метода `indexOf` используется в качестве оператора (в продолжение предыдущего примера):

```
scala> s indexOf ('o', 5) // Scala вызывает s.indexOf('o', 5)
res1: Int = 8
```

Оператором может быть любой метод

Операторы в Scala не являются специальным синтаксисом языка, поэтому оператором может быть любой метод. В оператор метод превращает способ его применения. Когда записывается код `s.indexOf('o')`, метод `indexOf` не является оператором. Но когда запись принимает вид `s indexOf 'o'`, метод `indexOf` становится оператором, поскольку при его использовании применяется система записи операторов.

До сих пор рассматривались только примеры *инфиксной* системы записи операторов, означающей, что вызываемый метод находится между объектом и параметром или параметрами, которые нужно передать методу, как в выражении `7 + 2`. В Scala также имеются две другие системы записи операторов: префиксная и постфиксная. В префиксной системе записи имя метода ставится перед объектом, в отношении которого этот метод вызывается (например, `-` в выражении `-7`). В постфиксной системе записи имя метода ставится после объекта (например, `toLong` в выражении

7 toLong).

В отличие от инфиксной системы записи операторов, в которой операторы получают два операнда, один слева, другой справа, префиксные и постфиксные операторы являются *унарными* — они получают только один операнд. В префиксной системе записи операнд размещается справа от оператора. В качестве примеров можно привести выражения `-2.0`, `!found` и `~0xFF`. Как и в случае использования инфиксных операторов, эти префиксные операторы являются сокращенной формой вызова методов. Но в данном случае перед символом оператора в имени метода ставится приставка `unary_`. Например, Scala превратит выражение `-2.0` в вызов метода `(2.0).unary_-`. Вы можете убедиться в этом, набрав вызов метода как с использованием системы записи операторов, так и в явном виде:

```
scala> -2.0 // Scala вызывает (2.0).unary_-
res2: Double = -2.0

scala> (2.0).unary_-
res3: Double = -2.0
```

Единственными идентификаторами, которые могут использоваться в качестве префиксных операторов, являются `+`, `-`, `!` и `~`. Следовательно, если вы определите метод по имени `unary_!`, то сможете вызвать его в отношении значения или переменной подходящего типа, применяя префиксную систему записи операторов, например `!p`. Но если вы определите метод по имени `unary_*`, то не сможете использовать префиксную систему записи операторов, поскольку `*` не входит в число четырех идентификаторов, которые могут задействоваться в качестве префиксных операторов. Метод можно вызвать обычным способом как `inp.unary_*`, но при попытке вызова его в виде `*p` Scala воспримет код, как будто он записан в виде `*.p`, что, вероятно,

совершенно не совпадает с задуманным[45]! Постфиксные операторы, когда они вызываются без точки или круглых скобок, являются методами, не получающими аргументов. В Scala при вызове метода пустые круглые скобки можно не ставить. Соглашение гласит, что круглые скобки добавляются в том случае, если метод имеет побочные эффекты, как в случае с методом `println()`. Но их можно не ставить, если метод не имеет побочных эффектов, как в случае с методом `toLowerCase`, вызываемым в отношении значения типа `String`:

```
scala> val s = "Hello, world!"
s: String = Hello, world!

scala> s.toLowerCase
res4: String = hello, world!
```

В последнем случае, где методу не требуются аргументы, можно при желании не ставить точку и воспользоваться постфиксной системой записи операторов:

```
scala> s toLowerCase
res5: String = hello, world!
```

Здесь метод `toLowerCase` применяется в качестве постфиксного оператора в отношении операнда `s`.

Чтобы понять, какие операторы можно использовать с основными типами Scala, нужно посмотреть на методы, объявленные в классах типов, в документации по Scala API. Но, поскольку данная книга является учебником по языку Scala, в нескольких следующих разделах будет дан краткий обзор большинства этих методов.

Ускоренный режим чтения для Java-программистов

Многие особенности Scala, рассматриваемые в оставшейся части главы, совпадают с аналогичными Java-особенностями. Если вы хорошо разбираетесь в Java и страдаете от недостатка времени, можете спокойно перейти к разделу 5.8, в котором описаны отличия Scala от Java в области равенства объектов.

5.5. Арифметические операции

Арифметические методы при работе с любыми числовыми типами можно вызвать в качестве инфиксных операторов для сложения (+), вычитания (-), умножения (*), деления (/) и получения остатка от деления (%). Вот несколько примеров:

```
scala> 1.2 + 2.3
res6: Double = 3.5
scala> 3 - 1
res7: Int = 2

scala> 'b' - 'a'
res8: Int = 1

scala> 2L * 3L
res9: Long = 6

scala> 11 / 4
res10: Int = 2

scala> 11% 4
res11: Int = 3

scala> 11.0f / 4.0f
res12: Float = 2.75
```

```
scala> 11.0% 4.0
res13: Double = 3.0
```

Когда целочисленными типами являются как правый, так и левый операнды (`Int`, `Long`, `Byte`, `Short` или `Char`), оператор `/` выведет всю числовую часть результата деления, исключая остаток. Остаток от предполагаемого целочисленного деления показывается с помощью оператора `%`.

Остаток от деления числа с плавающей точкой, полученный с помощью метода `%`, не определен в стандарте IEEE 754. Что касается операции вычисления остатка, в этом стандарте используется деление с округлением, а не деление с отбрасыванием остатка, поэтому данная операция сильно отличается от операции вычисления остатка от целочисленного деления. Если все-таки действительно нужно получить остаток по стандарту IEEE 754, можно вызвать метод `IEEEremainder` из `scala.math`:

```
scala> math.IEEEremainder(11.0, 4.0)
res14: Double = -1.0
```

Числовые типы также предлагают воспользоваться унарными префиксными операторами `+` (метод `unary_+`) и `-` (метод `unary_-`), позволяющими показать положительное или отрицательное значение числового литерала, как в `-3` или `+4.0`. Если не указать унарный `+` или `-`, число литерала интерпретируется как положительное. Унарный `+` существует исключительно для симметрии с унарным `-`, но не производит никаких действий. Унарный `-` может также использоваться для смены знака переменной. Вот несколько примеров:

```
scala> val neg = 1 + -3
neg: Int = -2
```

```
scala> val y = +3
y: Int = 3

scala> -neg
res15: Int = 2
```

5.6. Методы отношений и логические операции

Числовые типы можно сравнивать с помощью методов отношений «больше чем» (>), «меньше чем» (<), «больше чем или равно» (>=) и «меньше чем или равно» (<=), которые выдают в качестве результата булево значение. Дополнительно, чтобы инвертировать булево значение, можно использовать унарный оператор ! (метод unary_!). Вот несколько примеров:

```
scala> 1 > 2
res16: Boolean = false

scala> 1 < 2
res17: Boolean = true

scala> 1.0 <= 1.0
res18: Boolean = true

scala> 3.5f >= 3.6f
res19: Boolean = false

scala> 'a' >= 'A'
res20: Boolean = true

scala> val untrue = !true
```

```
untrue: Boolean = false
```

Методы «логическое И» (&& и &) и «логическое ИЛИ» (|| и |) получают операнды типа `Boolean` в инфиксной нотации и выдают результат в виде `Boolean`-значения, например:

```
scala> val toBe = true
toBe: Boolean = true

scala> val question = toBe || !toBe
question: Boolean = true

scala> val paradox = toBe && !toBe
paradox: Boolean = false
```

Операции && и ||, как и в Java, являются *сокращенно вычисляемыми*: выражения, построенные с использованием этих операторов, вычисляются только в том случае, если это нужно для определения результата. Иными словами, правая часть выражений с использованием && и || не будет вычисляться, если результат уже определился при решении левой части. Например, если левая часть выражения с методом && вычисляется в `false`, результатом выражения, несомненно, будет `false`, поэтому правая часть не вычисляется. Аналогично этому, если левая часть выражения с методом || вычисляется в `true`, результатом выражения, конечно же, будет `true`, поэтому правая часть не вычисляется:

```
scala> def salt() = { println("salt"); false }
salt: ()Boolean
scala> def pepper() = { println("pepper"); true }
pepper: ()Boolean

scala> pepper() && salt()
pepper
```

```
salt
res21: Boolean = false

scala> salt() && pepper()
salt
res22: Boolean = false
```

В первом выражении вызываются `pepper` и `salt`, но во втором — только `salt`. Поскольку `salt` возвращает `false`, необходимость в вызове `pepper` отпадает.

Если правую часть нужно вычислить при любых условиях, вместо прежних методов следует воспользоваться методами & и |. Метод & выполняет логическую операцию И, а метод | — операцию ИЛИ, но при этом они не прибегают к сокращенному вычислению, как это делают методы && и ||. Вот как выглядит пример их использования:

```
scala> salt() & pepper()
salt
pepper
res23: Boolean = false
```

ПРИМЕЧАНИЕ

Возникает вопрос: как может работать сокращенное вычисление, если операторы — это всего лишь методы? Обычно все аргументы вычисляются перед входом в метод, тогда каким же образом метод может избежать вычисления своего второго аргумента? Дело в том, что у всех методов Scala есть средство для задержки вычисления его аргументов или даже полной его отмены. Средство называется параметрами до востребования и будет рассмотрено в разделе 9.5.

5.7. Поразрядные операции

Scala позволяет выполнять операции над отдельными разрядами целочисленных типов, используя несколько поразрядных методов. К ним относятся поразрядное И (&), поразрядное ИЛИ (|) и поразрядное исключающее ИЛИ (^)[46]. Унарный поразрядный оператор дополнения (~, метод `unary_~`) инвертирует каждый разряд в своем операнде, например:

```
scala> 1 & 2
res24: Int = 0
scala> 1 | 2
res25: Int = 3

scala> 1 ^ 3
res26: Int = 2

scala> ~1
res27: Int = -2
```

В первом выражении, `1 & 2`, выполняется поразрядное И над каждым разрядом чисел 1 (`0001`) и 2 (`0010`) и выдается результат `0` (`0000`). Во втором выражении, `1 | 2`, выполняется поразрядное ИЛИ над теми же самыми операндами и выводится результат `3` (`0011`). В третьем выражении, `1 ^ 3`, выполняется поразрядное исключающее ИЛИ над каждым разрядом `1` (`0001`) и `3` (`0011`) и выдается результат `2` (`0010`). В последнем выражении, `~1`, инвертируется каждый разряд в `1` (`0001`) и выводится результат `-2`, который в двоичной форме выглядит как `11111111111111111111111111111110`.

Целочисленные типы Scala также предлагают три метода сдвига: сдвиг влево (<<), сдвиг вправо (>>) и сдвиг вправо без знака (>>>). Методы сдвига, примененные в инфиксной системе записи

операторов, сдвигают разряды целочисленного значения, указанные слева от оператора, на количество разрядов, заданное в целочисленном значении справа от оператора. При сдвиге влево и сдвиге вправо без знака разряды по мере сдвига заполняются нулями. При сдвиге вправо разряды указанного слева значения по мере сдвига заполняются значением самого старшего разряда (разряда знака). Вот несколько примеров:

```
scala> -1 >> 31
res28: Int = -1

scala> -1 >>> 31
res29: Int = 1

scala> 1 << 2
res30: Int = 4
```

Число –1 в двоичном виде выглядит как 11111111111111111111111111111111. В первом примере, `-1 >> 31`, в числе `-1` происходит сдвиг вправо на 31 разрядную позицию. Поскольку в значении типа `Int` содержатся 32 разряда, эта операция, по сути, перемещает самый левый разряд до тех пор, пока он не станет самым правым[47]. Поскольку метод >> выполняет заполнение по мере сдвига единицами, потому что самый левый разряд числа –1 — это 1, результат получается идентичным исходному левому операнду и состоит из 32 единичных разрядов или равняется –1. Во втором примере, `-1 >>> 31`, самый левый разряд опять сдвигается вправо в самую правую позицию, но на этот раз освобождающиеся разряды заполняются нулями. Поэтому результат в двоичном виде выглядит так: 00000000000000000000000000000001 или 1. В последнем примере, `1 << 2`, левый операнд `1` сдвигается влево на две позиции (освобождающиеся позиции заполняются нулями), в результате чего в двоичном виде получается число

00000000000000000000000000000100 или 4.

5.8. Одинаковое содержимое объектов

Если нужно сравнить два объекта на наличие одинакового содержимого, можно воспользоваться либо методом ==, либо его противоположностью — методом !=. Вот несколько простых примеров:

```
scala> 1 == 2
res31: Boolean = false

scala> 1 != 2
res32: Boolean = true

scala> 2 == 2
res33: Boolean = true
```

По сути, эти две операции применимы ко всем объектам, а не только к основным типам. Например, оператор == можно использовать для сравнения списков:

```
scala> List(1, 2, 3) == List(1, 2, 3)
res34: Boolean = true

scala> List(1, 2, 3) == List(4, 5, 6)
res35: Boolean = false
```

Если пойти еще дальше, можно сравнить два объекта, имеющих разные типы:

```
scala> 1 == 1.0
res36: Boolean = true
```

```
scala> List(1, 2, 3) == "hello"
res37: Boolean = false
```

Допустимо даже выполнить сравнение со значением `null` или с тем, что может иметь значение `null`. Никакие исключения при этом выдаваться не будут:

```
scala> List(1, 2, 3) == null
res38: Boolean = false

scala> null == List(1, 2, 3)
res39: Boolean = false
```

Как видите, оператор `==` реализован весьма искусно и вы в большинстве случаев получите то сравнение на наличие одинакового содержимого, которое вам нужно. Все делается по очень простому правилу: сначала левая часть проверяется на `null`. Если ее значение не `null`, вызывается метод `equals`. Поскольку `equals` является методом, точность получаемого сравнения зависит от типа левого аргумента. Так как проверка на `null` выполняется автоматически, вам ее делать не приходится[48].

Этот вид сравнения выдает `true` в отношении различных объектов, если их содержимое одинаково и их методы `equals` созданы на основе проверки содержимого. Например, вот как сравниваются две строки, в которых по пять одинаковых букв:

```
scala> ("he" + "llo") == "hello"
res40: Boolean = true
```

Чем оператор == в Scala отличается от аналогичного оператора в Java

В Java оператор == может использоваться для сравнения как

элементарных, так и ссылочных типов. В отношении элементарных типов оператор == в Java проверяет равенство значений, как и в Scala. Но для ссылочных типов оператор == в Java проверяет равенство ссылок. Это означает, что две переменные указывают на один и тот же объект в куче, принадлежащей JVM. В Scala также предоставляется средство `eq` для сравнения равенства ссылок. Но метод `eq` и его противоположность, метод `ne`, применяются только к объектам, которые непосредственно отображаются на объекты Java. Исчерпывающие подробности о `eq` и `ne` приводятся в разделах 11.1 и 11.2. Также в главе 30 показано, как можно создать весьма качественный метод `equals`.

5.9. Приоритетность и ассоциативность операторов

Приоритетность операторов определяет, какая часть выражения рассчитывается самой первой. Например, выражение `2 + 2 * 7` вычисляется в 16, а не в 28, поскольку оператор `*` имеет более высокий приоритет, чем оператор `+`. Поэтому та часть выражения, где требуется перемножить числа, вычисляется до того, как будет выполнена та часть, где числа складываются. Разумеется, чтобы уточнить в выражении порядок вычисления или переопределить приоритеты, можно воспользоваться круглыми скобками. Например, если вы действительно хотите, чтобы результат ранее показанного выражения был 28, можете набрать следующее выражение:

```
(2 + 2) * 7
```

Если учесть, что в Scala, по сути, нет операторов, а есть только способ использования методов в системе записи операторов,

возникает вопрос: а как тогда работает приоритетность операторов? Scala принимает решение о приоритетности на основе первого символа метода, использованного в системе записи операторов (из этого правила есть одно исключение, рассматриваемое на следующих страницах). Если имя метода начинается, к примеру, с *, он получит более высокий приоритет, чем метод, чье имя начинается на +. Следовательно, выражение `2 + 2 * 7` будет вычислено как `2 + (2 * 7)`. Аналогично этому выражение `a +++ b *** c`, в котором `a`, `b` и `c` являются переменными, а +++ и *** — методами, будет вычислено как `a +++ (b *** c)`, поскольку метод *** обладает более высоким уровнем приоритета, чем метод +++.

1. Все специальные символы.
2. * / %
3. + –
4. :
5. = !
6. < >
7. &
8. ^
9. |
10. Все буквы.
11. Все операторы присваивания.

В этом списке показана приоритетность применительно к

первому символу метода в убывающем порядке, где символы, расположенные на одной строке, определяют одинаковый уровень приоритета. Чем выше символ в списке, тем выше приоритет начинающегося с него метода. Вот пример, показывающий влияние приоритетности:

```
scala> 2 << 2 + 2
res41: Int = 32
```

Имя метода << начинается с символа <, появляющегося в приведенном списке ниже символа +, который является первым и единственным символом метода +. Следовательно, << будет иметь более низкий уровень приоритета, чем +, и выражение будет рассчитано путем вызова сначала метода +, а затем метода <<, как в выражении 2 << (2 + 2). При сложении 2 + 2 в результате математического действия получается 4, а вычисление выражения 2 << 4 дает результат 32. Если поменять операторы местами, получится другой результат:

```
scala> 2 + 2 << 2
res42: Int = 16
```

Поскольку первые символы по сравнению с предыдущим примером не изменились, методы будут вызваны в том же порядке: сначала метод +, а затем метод <<. Следовательно, при 2 + 2 опять получится 4, а 4 << 2 даст результат 16.

Единственное исключение из правил, о существовании которого уже говорилось, относится к *операторам присваивания*, заканчивающимся знаком равенства. Если оператор заканчивается знаком равенства (=) и не относится к одному из операторов сравнения <=, >=, == и !=, приоритетность оператора имеет такой же уровень, что и у простого присваивания (=). То есть она ниже приоритетности любого другого оператора. Например, код:

```
x *= y + 1
```

означает то же самое, что и:

```
x *= (y + 1)
```

поскольку оператор *= классифицируется как оператор присваивания, чья приоритетность ниже, чем у +, даже при том, что первым символом оператора является знак *, который обозначил бы приоритетность выше, чем у +.

Если в выражении рядом появляются операторы с одинаковым уровнем приоритета, способ группировки операторов устанавливается их *ассоциативностью*. Ассоциативность оператора в Scala определяется его последним символом. Как уже упоминалось в главе 3, любой метод, имя которого оканчивается знаком :, вызывается в отношении своего правого операнда с передачей ему левого операнда. Методы, в окончании имени которых используются любые другие символы, действуют наоборот: они вызываются в отношении своего левого операнда с передачей этим методам правого операнда. Следовательно, из выражения `a * b` получается `a.*(b)`, но из `a ::: b` получается `b.:::(a)`.

Но независимо от того, какой ассоциативностью обладает оператор, его операнды всегда вычисляются слева направо. Следовательно, если `a` — выражение, не являющееся простой ссылкой на неизменяемое значение, то выражение `a ::: b` при более точном рассмотрении представляется следующим блоком:

```
{ val x = a; b.:::(x) }
```

В этом блоке `a` по-прежнему вычисляется раньше `b`, а затем результат этого вычисления передается в качестве операнда принадлежащему `b` методу `:::`.

Это правило ассоциативности играет роль также при появлении в одном выражении рядом сразу нескольких операторов с одинаковым уровнем приоритета. Если имена методов

оканчиваются на `:`, они группируются справа налево, в противном случае — слева направо. Например, `a ::: b ::: c` рассматривается как `a ::: (b ::: c)`. Но `a * b * c`, в отличие от этого, представляется как `(a * b) * c`.

Приоритетность операторов является частью языка Scala, и вам не следует бояться ею пользоваться. При этом, чтобы прояснить первоочередность применения операторов, в некоторых выражениях все же лучше воспользоваться круглыми скобками. Пожалуй, единственное, на что можно реально рассчитывать в отношении знания порядка приоритетности другими программистами, — это то, что мультипликативные операторы `*`, `/` и `%` имеют более высокий уровень приоритета, чем аддитивные `+` и `-`. Таким образом, даже если выражение `a + b << c` выдает нужный результат и без круглых скобок, внесение дополнительной ясности, которую можно обеспечить, используя запись `(a + b) << c`, снизит количество нелестных отзывов ваших коллег по поводу задействованной вами системы записи операторов, что выражается, к примеру, в недовольном восклицании вроде «опять в его коде невозможно разобраться!» и отправке вам сообщения вроде `bills !*&^%~ code!`[49].

5.10. Обогащающие оболочки

В отношении основных типов Scala можно вызвать намного больше методов, чем рассмотрено в предыдущих разделах. Некоторые примеры показаны в табл. 5.3. Эти методы становятся доступными при использовании *подразумеваемого преобразования* — технологии, подробное описание которой будет дано в главе 21. А пока вам нужно лишь знать, что для каждого основного типа, рассмотренного в данной главе, существует обогащающая оболочка, предоставляющая ряд дополнительных методов. Поэтому, чтобы увидеть все доступные методы, применяемые в отношении основных типов, нужно обратиться к документации по

API, касающейся обогащающей оболочки для каждого основного типа. Классы, составляющие обогащающие оболочки, перечислены в табл. 5.4.

Таблица 5.3. Некоторые обогащающие операции

Код	Результат
0 max 5	5
0 min 5	0
–2.7 abs	2.7
–2.7 round	–3L
1.5 isInfinity	False
(1.0 / 0) isInfinity	True
4 to 6	Range(4, 5, 6)
"bob" capitalize	"Bob"
"robert" drop 2	"bert"

Таблица 5.4. Классы обогащающих оболочек

Основной тип	Обогащающая оболочка
Byte	scala.runtime.RichByte
Short	scala.runtime.RichShort
Int	scala.runtime.RichInt
Long	scala.runtime.RichLong
Char	scala.runtime.RichChar
Float	scala.runtime.RichFloat
Double	scala.runtime.RichDouble
Boolean	scala.runtime.RichBoolean
String	scala.collection.immutable.StringOps

Резюме

Основное, что следует усвоить, прочитав эту главу, — это то, что операторы в Scala являются вызовами методов и для основных типов Scala существует подразумеваемое преобразование в обогащенные варианты, добавляющее дополнительные полезные методы. В следующей главе будет показано, что означает конструирование объектов в функциональном стиле, обеспечивающее новые реализации некоторых операторов, рассмотренных в данной главе.

43 Пакеты, кратко рассмотренные в шаге 1 главы 2, более подробно описаны в главе 13.

44 Перегружаемые методы имеют точно такие же имена, но используют другие типы аргументов. Более подробно перегрузка методов рассматривается в разделе 6.11.

45 Но не обязательно все будет потеряно. Есть весьма незначительная вероятность того, что программа с кодом *p может откомпилироваться как код C++.

46 Метод поразрядного исключающего ИЛИ выполняет соответствующую операцию в отношении своих операндов. Из одинаковых разрядов получается 0, а из разных — 1. Следовательно, выражение 0011 ^ 0101 вычисляется в 0110.

47 Самый левый разряд в целочисленном типе является знаковым разрядом. Если самый левый разряд установлен в 1, значит число отрицательное. Когда он установлен в 0 —число положительное.

48 Автоматическая проверка игнорирует правую сторону, но любой корректно реализованный метод equals должен возвращать false, если его аргумент имеет значение null.

49 Теперь вы уже знаете, что, получив такой код, компилятор Scala создаст вызов (bills.!*&^%~(code)).!().

6. Функциональные объекты

Усвоив основы, рассмотренные в предыдущих главах, вы готовы к разработке более полнофункциональных классов Scala. В этой главе основное внимание будет уделено классам, определяющим функциональные объекты, или объекты, у которых нет никакого изменяемого состояния. Запуская примеры, мы создадим несколько вариантов класса, моделирующего рациональные числа в виде неизменяемых объектов. Попутно будут показаны дополнительные аспекты объектно-ориентированного программирования на Scala: параметры класса и конструкторы, методы и операторы, закрытые элементы, переопределение, проверка соблюдения предварительных условий, перегрузка и рекурсивные ссылки.

6.1. Спецификация класса Rational

Рациональным называется число, которое может быть выражено соотношением `n/d`, где `n` и `d` представлены целыми числами, за исключением того, что `d` не может быть нулем. Здесь `n` называется числителем, а `d` — знаменателем. Примерами рациональных чисел могут послужить 1/2, 2/3, 112/239 и 2/1. В сравнении с числами с плавающей точкой рациональные числа имеют то преимущество, что дроби представлены точно, без округлений или приближений.

Разрабатываемый в этой главе класс должен моделировать поведение рациональных чисел, позволяя производить над ними действия по сложению, вычитанию, умножению и делению. Для сложения двух рациональных чисел сначала нужно получить общий знаменатель, после чего сложить два числителя. Например, чтобы выполнить сложение 1/2 + 2/3, обе части левого операнда умножаются на 3, а обе части правого операнда — на 2, в результате чего получается 3/6 + 4/6. Сложение двух числителей дает результат 7/6. Для перемножения двух рациональных чисел можно просто

перемножить их числители и их знаменатели. Таким образом, 1/2 × 2/5 дает число 2/10, которое может быть представлено более кратко в нормализованном виде как 1/5. Деление выполняется путем перестановки местами числителя и знаменателя правого операнда с последующим перемножением чисел. Например, 1/2 / 3/5 — это то же самое, что и 1/2 × 5/3, в обоих случаях в результате получается число 5/6.

Очевидное, казалось бы, наблюдение заключается в том, что в математике рациональные числа не имеют изменяемого состояния. Одно рациональное число можно прибавить к другому, и результатом будет новое рациональное число. Исходные числа не изменятся. Неизменяемый класс `Rational`, разрабатываемый в данной главе, будет обладать таким же свойством. Каждое рациональное число будет представлено одним `Rational`-объектом. При сложении двух `Rational`-объектов для хранения суммы создается новый `Rational`-объект.

В этой главе мы рассмотрим некоторые допустимые в Scala способы написания библиотек, создающих такое впечатление, будто используется поддержка, присущая непосредственно самому языку программирования. Например, в конце этой главы вы сможете сделать с классом `Rational` следующее:

```
scala> val oneHalf = new Rational(1, 2)
oneHalf: Rational = 1/2

scala> val twoThirds = new Rational(2, 3)
twoThirds: Rational = 2/3

scala> (oneHalf / 7) + (1 - twoThirds)
res0: Rational = 17/42
```

6.2. Конструкция класса Rational

Конструирование класса `Rational` неплохо начать с рассмотрения того, как клиенты-программисты будут создавать новый `Rational`-объект. Поскольку решено было делать `Rational`-объекты неизменяемыми, мы потребуем, чтобы эти клиенты при создании экземпляра обеспечивали все необходимые ему данные (в нашем случае числитель и знаменатель). Поэтому начнем конструкцию со следующего кода:

```
class Rational(n: Int, d: Int)
```

По поводу этой строки кода в первую очередь следует заметить, что, если у класса нет тела, вам не нужно ставить пустые фигурные скобки (конечно, если хотите, можете поставить). Идентификаторы `n` и `d`, указанные в круглых скобках после имени класса, `Rational`, называются параметрами класса. Компилятор Scala подберет эти два параметра класса и создаст первичный конструктор, получающий эти же два параметра.

Плюсы и минусы неизменяемого объекта

Неизменяемые объекты имеют ряд преимуществ перед изменяемыми и один потенциальный недостаток. Во-первых, о неизменяемых объектах проще говорить, чем о неизменяемых, поскольку у них нет изменяемых со временем сложных областей состояния. Во-вторых, неизменяемые объекты можно совершенно свободно куда-нибудь передавать, а перед передачей изменяемых объектов в другой код порой приходится делать страховочные копии. В-третьих, если объект правильно сконструирован, то при одновременном обращении к неизменяемому объекту из двух потоков повредить его состояние невозможно, поскольку никакой поток не может изменить состояние

неизменяемого объекта. В-четвертых, неизменяемые объекты обеспечивают безопасность ключей хеш-таблиц. Если, к примеру, изменяемый объект поменялся после помещения его в HashSet, в следующий раз при поиске в HashSet его можно не найти.

Главным недостатком неизменяемых объектов является то, что им иногда требуется копирование больших графов объектов, в то время как обновление может быть внесено на месте. В некоторых случаях могут возникать затруднения в выражении, также возможны узкие места, снижающие производительность. В результате в библиотеки нередко включают изменяемые альтернативы неизменяемым классам. Например, класс `StringBuilder` является изменяемой альтернативой неизменяемого класса `String`. Дополнительная информация о конструировании изменяемых объектов в Scala будет дана в главе 18.

ПРИМЕЧАНИЕ

В исходном примере с Rational выявляется разница между Java и Scala. В Java классы обладают конструкторами, которые могут принимать параметры, а в Scala классы могут принимать параметры напрямую. Система записи в Scala значительно лаконичнее — параметры класса могут использоваться напрямую в теле, нет никакой необходимости определять поля и записывать присваивания, копирующие параметры конструктора в поля. Это может привести к дополнительной экономии на шаблонном коде, особенно когда дело касается небольших классов.

Компилятор откомпилирует любой код, помещенный в тело

класса и не являющийся частью поля или определения метода, в первичный конструктор. Например, можно вывести следующее отладочное сообщение:

```
class Rational(n: Int, d: Int) {
  println("Created " + n + "/" + d)
}
```

Получив этот код, компилятор Scala поместит вызов `println` в первичный конструктор класса `Rational`. Поэтому при создании нового экземпляра `Rational` вызов `println` приведет к выводу отладочного сообщения:

```
scala> new Rational(1, 2)
Created 1/2
res0: Rational = Rational@2591e0c9
```

6.3. Переопределение метода toString

При создании экземпляра `Rational` в предыдущем примере интерпретатор вывел `Rational@90110a`. Эта странная строка получилась путем вызова в отношении `Rational`-объекта метода `toString`. По умолчанию класс `Rational` наследует реализацию `toString`, определенную в классе `java.lang.Object`, которая просто выводит имя класса, символ `@` и шестнадцатеричное число. Предполагалось, что результат выполнения `toString` поможет программистам, предоставив информацию, которая может быть использована при отладке инструкций вывода информации, регистрационных сообщений, отчетов о сбоях тестов, а также выходных сведений интерпретатора и отладчика. Результат, выдаваемый на данный момент методом `toString`, не приносит особой пользы, поскольку не дает никакой информации относительно значения рационального числа. Более полезная реализация `toString` будет выводить значения числителя и

знаменателя Rational-объекта. Переопределить исходную реализацию можно добавлением метода toString к классу Rational:

```
class Rational(n: Int, d: Int) {
  override def toString = n + "/" + d
}
```

Модификатор override перед определением метода показывает, что предыдущее определение метода переустанавливается (более подробно этот вопрос рассматривается в главе 10). Поскольку теперь числа типа Rational будут выводиться совершенно отчетливо, мы удаляем отладочную инструкцию println, помещенную в тело предыдущей версии класса Rational. В итоге новое поведение Rational можно протестировать в интерпретаторе:

```
scala> val x = new Rational(1, 3)
x: Rational = 1/3

scala> val y = new Rational(5, 7)
y: Rational = 5/7
```

6.4. Проверка соблюдения предварительных условий

В качестве следующего шага переключим внимание на проблему, связанную с текущим поведением первичного конструктора. Как упоминалось в начале главы, рациональные числа не должны содержать нуль в знаменателе. Но пока первичный конструктор принимает нуль, передаваемый в качестве параметра d:

```
scala> new Rational(5, 0)
res1: Rational = 5/0
```

Одним из преимуществ объектно-ориентированного программирования является возможность инкапсуляции данных внутри объектов, чтобы можно было гарантировать, что данные приемлемы в течение всей жизни объекта. В нашем случае для такого неизменяемого объекта, как `Rational`, это означает, что вы должны гарантировать приемлемость данных на этапе конструирования объекта при условии, что нулевой знаменатель является недопустимым состоянием числа типа `Rational` и такое число не должно создаваться, если в качестве параметра `d` передается нуль.

Лучше всего решить эту проблему, определив для первичного конструктора предусловие, согласно которому `d` должен иметь ненулевое значение. Предусловие представляет собой ограничение, накладываемое на значения, передаваемые в метод или конструктор, то есть требование, которое должно выполняться вызывающим кодом. Одним из способов решения задачи является использование метода `require`[50]:

```
class Rational(n: Int, d: Int) {
  require(d != 0)
  override def toString = n + "/" + d
}
```

Метод `require` получает один булев параметр. Если переданное значение приведет к вычислению в `true`, из метода `require` произойдет нормальный выход. В противном случае объект не будет создан и появится исключение `IllegalArgumentException`.

6.5. Добавление полей

Теперь, когда первичный конструктор выдвигает нужные предусловия, мы переключимся на поддержку сложения. Для этого

определим в классе Rational открытый метод add, получающий в качестве параметра еще одно значение типа Rational. Чтобы сохранить неизменяемость класса Rational, метод add не должен прибавлять переданное рациональное число к объекту, в отношении которого он вызван. Ему нужно создать и вернуть новый Rational-объект, содержащий сумму. Можно подумать, что метод add создается следующим образом:

```
class Rational(n: Int, d: Int) { // Этот код не будет откомпилирован
  require(d != 0)
  override def toString = n + "/" + d
  def add(that: Rational): Rational =
      new Rational(n * that.d + that.n * d, d * that.d)
}
```

Но, получив этот код, компилятор выдаст свои возражения:

```
<console>:11: error: value d is not a member of Rational
           new Rational(n * that.d + that.n * d, d * that.d)
                                        ^
<console>:11: error: value d is not a member of Rational
           new Rational(n * that.d + that.n * d, d * that.d)
```

Хотя параметры класса n и d находятся в области видимости кода вашего метода add, получить доступ к их значению можно только в объекте, в отношении которого вызван метод add. Следовательно, когда в реализации метода add указывается n или

d, компилятор готов предоставить вам значения для этих параметров класса. Но он не может позволить указать that.n или that.d, поскольку они не ссылаются на Rational-объект, в отношении которого был сделан вызов метода add[51]. Чтобы получить таким образом доступ к числителю или знаменателю, их нужно сделать полями. Как эти поля можно добавить к классу Rational, показано в листинге 6.1[52].

Листинг 6.1. Класс Rational с полями

```
class Rational(n: Int, d: Int) {
  require(d != 0)
  val numer: Int = n
  val denom: Int = d
  override def toString = numer + "/" + denom
  def add(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
}
```

В версии Rational, показанной в листинге 6.1, добавлены два поля с именами numer и denom, которые были проинициализированы значениями параметров класса n и d[53]. Также были внесены изменения в реализацию методов toString и add, позволяющие им использовать поля, а не параметры класса. Эта версия класса Rational проходит компиляцию. Ее можно протестировать путем сложения рациональных чисел:

```
scala> val oneHalf = new Rational(1, 2)
oneHalf: Rational = 1/2
```

```
scala> val twoThirds = new Rational(2, 3)
twoThirds: Rational = 2/3

scala> oneHalf add twoThirds
res2: Rational = 7/6
```

Теперь вы уже можете сделать то, чего не могли сделать раньше, а именно получить доступ к значениям числителя и знаменателя из-за пределов объекта. Для этого нужно просто обратиться к открытым полям numer и denom:

```
scala> val r = new Rational(1, 2)
r: Rational = 1/2

scala> r.numer
res3: Int = 1

scala> r.denom
res4: Int = 2
```

6.6. Рекурсивные ссылки

Ключевое слово this позволяет сослаться на экземпляр объекта, в отношении которого был вызван выполняемый в данный момент метод, или, если оно использовалось в конструкторе, — на создаваемый экземпляр объекта. Рассмотрим в качестве примера добавление метода lessThan, проверяющего, не имеет ли Rational-объект, в отношении которого он вызван, значение, меньшее значения параметра:

```
def lessThan(that: Rational) =
     this.numer * that.denom < that.numer *
this.denom
```

Здесь выражение `this.numer` ссылается на числительное объекта, в отношении которого вызван метод `lessThan`. Можно также не указывать префикс `this` и написать просто `numer`, обе записи будут равнозначны.

В качестве примера случаев, когда вам не обойтись без `this`, рассмотрим добавление к классу `Rational` метода `max`, возвращающего наибольшее число из заданного рационального числа и переданного аргумента:

```
def max(that: Rational) =
  if (this.lessThan(that)) that else this
```

Здесь первое ключевое слово `this` является избыточным. Можно его не указывать и написать `lessThan(that)`. Но второе ключевое слово `this` представляет результат метода в том случае, когда тест вернет `false`, и, если вы его не укажете, возвращать будет просто нечего!

6.7. Дополнительные конструкторы

Иногда нужно, чтобы в классе было несколько конструкторов. В Scala все конструкторы, кроме первичного, называются дополнительными. Например, рациональное число со знаменателем 1 может быть кратко записано в виде числителя. Например, вместо 5/1 можно просто указать 5. Поэтому было бы неплохо, чтобы вместо записи `new Rational(5, 1)` программисты-клиенты могли написать лишь `new Rational(5)`. Для этого потребуется добавить к классу `Rational` дополнительный конструктор, получающий только один аргумент — числитель, а в качестве знаменателя имеющий предопределенное значение 1. Как может выглядеть соответствующий код, показано в листинге 6.2.

Листинг 6.2. Класс Rational с дополнительным конструктором

```
class Rational(n: Int, d: Int) {

require(d != 0)

val numer: Int = n
val denom: Int = d
def this(n: Int) = this(n, 1) // дополнительный
конструктор

override def toString = numer + "/" + denom

def add(that: Rational): Rational =
  new Rational(
    numer * that.denom + that.numer * denom,
    denom * that.denom
  )
}
```

Определения дополнительных конструкторов в Scala начинаются с `def this(...)`. Тело дополнительного конструктора класса `Rational` просто вызывает первичный конструктор, передавая дальше свой единственный аргумент `n` в качестве числителя и 1 — в качестве знаменателя. Увидеть дополнительный конструктор в действии можно, набрав в интерпретаторе следующий код:

```
scala> val y = new Rational(3)
y: Rational = 3/1
```

В Scala каждый дополнительный конструктор первым действием должен вызывать еще один конструктор того же класса. Иными словами, первая инструкция в каждом дополнительном

конструкторе каждого класса Scala должна иметь вид `this(...)`. Вызываемым конструктором должен быть либо первичный конструктор (как в примере с классом `Rational`), либо другой дополнительный конструктор, который появляется в тексте программы перед вызывающим его конструктором. Конечный результат применения данного правила заключается в том, что каждый вызов конструктора в Scala обязан в конце концов завершаться вызовом первичного конструктора класса. Первичный конструктор, таким образом, является единственной точкой входа в класс.

ПРИМЕЧАНИЕ

Знатоков Java может удивить то, что в Scala правила в отношении конструкторов более строгие, чем в Java. Ведь в Java первым действием конструктора должен быть либо вызов другого конструктора того же класса, либо вызов конструктора родительского класса напрямую. В классе Scala конструктор родительского класса может быть вызван только первичным конструктором. Более сильные ограничения в Scala фактически являются конструкторским компромиссом — платой за большую лаконичность и простоту конструкторов Scala по сравнению с конструкторами Java. Родительские классы и подробности вызова конструктора и наследования будут рассмотрены в главе 10.

6.8. Закрытые поля и методы

В предыдущей версии класса `Rational` мы просто инициализировали `numer` значением `n`, а `denom` — значением `d`. Из-за этого числитель и знаменатель `Rational` могут превышать необходимые значения. Например, дробь 66/42 можно сократить и привести к виду 11/7, но первичный конструктор класса `Rational`

пока этого не делает:

```
scala> new Rational(66, 42)
res5: Rational = 66/42
```

Чтобы выполнить такое сокращение, нужно разделить числитель и знаменатель на их наибольший общий делитель. Например, наибольшим общим делителем для 66 и 42 будет число 6. (Иными словами, 6 является наибольшим целым числом, на которое без остатка делится как 66, так и 42.) Деление и числителя, и знаменателя числа 66/42 на 6 приводит к получению сокращенной формы 11/7. Один из способов решения данной задачи показан в листинге 6.3.

Листинг 6.3. Класс Rational с закрытым полем и методом

```
class Rational(n: Int, d: Int) {

  require(d != 0)

  private val g = gcd(n.abs, d.abs)
  val numer = n / g
  val denom = d / g
  def this(n: Int) = this(n, 1)

  def add(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  override def toString = numer + "/" + denom
```

```
  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
}
```

В данной версии класса `Rational` было добавлено закрытое поле `g` и изменены инициализаторы для полей `numer` и `denom`. (Инициализатором называется код, вводящий в действие переменную, например `n / g`, который инициализирует поле `numer`.) Поскольку поле `g` является закрытым, доступ к нему может быть выполнен изнутри, но не снаружи тела класса.

Также был добавлен закрытый метод по имени `gcd`, вычисляющий наибольший общий делитель двух переданных ему `Int`-значений. Например, вызов `gcd(12, 8)` дает результат 4. Как было показано в разделе 4.1, чтобы сделать поле или метод закрытым, следует просто поставить перед его определением ключевое слово `private`. Назначением закрытого вспомогательного метода `gcd` является обособление кода, необходимого для остальных частей класса, в данном случае для первичного конструктора. Чтобы обеспечить постоянное положительное значение поля `g`, методу передаются абсолютные значения параметров `n` и `d`, получаемые вызовом в отношении этих параметров метода `abs`, который может вызываться в отношении любого `Int`-объекта для получения его абсолютного значения.

Компилятор Scala поместит коды инициализаторов трех полей класса `Rational` в первичный конструктор в порядке их следования в исходном коде. Таким образом, инициализатор поля `g`, имеющий код `gcd(n.abs, d.abs)`, будет выполнен до двух других инициализаторов, поскольку в исходном коде он появляется первым. Поле `g` будет инициализировано результатом — наибольшим общим делителем абсолютных значений параметров класса `n` и `d`. Затем поле `g` задействуется в инициализаторах полей `numer` и `denom`. Путем деления `n` и `d` на их

наибольший общий делитель `g` каждый `Rational`-объект будет сконструирован в нормализованной форме:

```
scala> new Rational(66, 42)
res6: Rational = 11/7
```

6.9. Определение операторов

Текущая реализация класса `Rational` нас вполне устраивает, но ее можно сделать гораздо удобнее в использовании. Зададимся вопросом, почему можно воспользоваться записью

```
x + y
```

если `x` и `y` являются целыми числами или числами с плавающей точкой. Но когда это рациональные числа, то приходится пользоваться записью:

```
x.add(y)
```

или в крайнем случае:

```
x add y
```

Такое положение дел ничем не оправдано. Рациональные числа не отличаются от остальных чисел. В математическом смысле они гораздо естественнее, скажем, чисел с плавающей точкой. Так почему бы не воспользоваться во время работы с ними естественными математическими операторами? И в Scala есть такая возможность. Как этот делается, будет показано в оставшейся части главы.

Сначала нужно заменить `add` обычным математическим символом. Сделать это нетрудно, поскольку знак + является в Scala вполне допустимым идентификатором. Можно просто определить метод с именем +. Если уж на то пошло, можно также определить и

метод *, выполняющий умножение. Результат показан в листинге 6.4.

Листинг 6.4. Класс Rational с методами-операторами

```
class Rational(n: Int, d: Int) {

  require(d != 0)

  private val g = gcd(n.abs, d.abs)
  val numer = n / g
  val denom = d / g
  def this(n: Int) = this(n, 1)

  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  def * (that: Rational): Rational =
       new Rational(numer * that.numer, denom *
that.denom)

  override def toString = numer + "/" + denom

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
}
```

После такого определения класса Rational можно будет воспользоваться следующим кодом:

```
scala> val x = new Rational(1, 2)
x: Rational = 1/2

scala> val y = new Rational(2, 3)
y: Rational = 2/3

scala> x + y
res7: Rational = 7/6
```

Как всегда, синтаксис оператора в последней строке ввода является эквивалентом вызова метода. Можно также использовать следующий код:

```
scala> x.+(y)
res8: Rational = 7/6
```

но читать его будет намного труднее.

Следует также заметить, что из-за действующих в Scala правил приоритетности операторов, рассмотренных в разделе 5.9, метод * будет привязан к Rational-объектам сильнее метода +. Иными словами, выражения, в которых в качестве Rational-объектов используются операции + и *, поведут себя вполне ожидаемым образом. Например, `x + x * y` будет выполняться как `x + (x * y)`, а не как `(x + x) * y`:

```
scala> x + x * y
res9: Rational = 5/6

scala> (x + x) * y
res10: Rational = 2/3

scala> x + (x * y)
res11: Rational = 5/6
```

6.10. Идентификаторы в Scala

Вам уже встречались два наиболее важных способа составления идентификаторов в Scala — из буквенно-цифровых символов и из операторов. В Scala используются весьма гибкие правила формирования идентификаторов. Кроме двух уже встречавшихся форм, существуют еще две формы. Все четыре формы составления идентификаторов рассматриваются в этом разделе.

Буквенно-цифровые идентификаторы начинаются с буквы или знака подчеркивания, за которыми могут следовать другие буквы, цифры или знаки подчеркивания. Символ `$` также считается буквой, но он зарезервирован для идентификаторов, создаваемых компилятором Scala. Идентификаторы в пользовательских программах не должны содержать символы `$`, несмотря на то что они могут успешно пройти компиляцию: если это произойдет, возможны конфликты имен с теми идентификаторами, которые будут созданы компилятором Scala.

В Scala соблюдается соглашение, принятое в Java относительно применения идентификаторов в смешанном регистре[54], таких как `toString` и `HashSet`. Хотя использование знаков подчеркивания в идентификаторах вполне допустимо, в программах на Scala они встречаются довольно редко — отчасти для соблюдения совместимости с Java, а также из-за того, что знаки подчеркивания в коде Scala активно применяются не только для идентификаторов. Поэтому лучше избегать таких идентификаторов, как, например, `to_string`, `__init__` или `name_`. Имена полей, параметры методов, имена локальных переменных и имена функций в смешанном регистре должны начинаться с буквы в нижнем регистре, например: `length`, `flatMap` и `s`. Имена классов и трейтов в смешанном регистре должны начинаться с буквы в верхнем регистре, например: `BigInt`, `List` и `UnbalancedTreeMap`[55].

ПРИМЕЧАНИЕ

Одним из последствий использования в идентификаторе замыкающего знака подчеркивания при попытке, к примеру, написания объявления val name_: Int = 1, может стать ошибка компиляции. Компилятор подумает, что вы пытаетесь объявить val-переменную по имени name_:. Чтобы такой идентификатор прошел компиляцию, перед двоеточием нужно поставить дополнительный пробел, как в коде val name_ : Int = 1.

Один из примеров отступления Scala от соглашений, принятых в Java, касается имен констант. В Scala слово «константа» не означается просто `val`-переменную. Даже при том что `val`-переменная остается неизменной после инициализации, она не перестает быть переменной. Например, параметры метода относятся к `val`-переменным, но при каждом вызове метода в этих `val`-переменных содержатся разные значения. Константа обладает более выраженным постоянством. Например, `scala.math.Pi` определяется как значение с двойной точностью, наиболее близкое к реальному значению числа π — отношению длины окружности к ее диаметру. Это значение вряд ли когда-то изменится, поэтому со всей очевидностью можно сказать, что `Pi` является константой. Константы можно использовать также для присваивания имен значениям, которые иначе были бы в вашем коде непонятными числами — буквальными значениями без объяснений, которые в худшем случае появлялись бы в коде в нескольких местах. Может также понадобиться определить константы для использования при определении соответствий тем или иным шаблонам, что будет рассматриваться в разделе 15.2. В соответствии с соглашением, принятым в Java, константам присваиваются имена с использованием символов в верхнем регистре, где знак подчеркивания является разделителем слов, например `MAX_VALUE` или `PI`. В Scala соглашение требует, чтобы в верхнем регистре была только первая буква. Таким образом, константы, названные в стиле

Java, например `X_OFFSET`, будут работать в Scala в качестве констант, но по соглашению, принятому в Scala, для имен констант применяется смешанный регистр, например `XOffset`.

Идентификатор оператора состоит из одного или нескольких символов операторов. Символами операторов являются выводимые на печать ASCII-символы, например `+`, `:`, `?`, `~` или `#`[56]. Вот как выглядят некоторые примеры идентификаторов операторов:

```
+ ++ ::: <?> :->
```

Компилятор Scala на внутреннем уровне перерабатывает идентификаторы операторов, чтобы превратить их в допустимые Java-идентификаторы со встроенными символами `$`. Например, идентификатор `:->` будет представлен как `$colon$minus$greater`. Если вам когда-либо захочется получить доступ к этому идентификатору из кода Java, то для этого потребуется использовать данное внутреннее представление.

Поскольку идентификаторы операторов в Scala могут принимать произвольную длину, между Java и Scala есть небольшая разница в этом вопросе. В Java введенный код `x<-y` будет разобран на четыре лексических символа, в результате чего он станет эквивалентен `x < - y`. В Scala оператор `<-` будет рассмотрен как единый идентификатор, в результате чего получится `x <- y`. Если нужно получить первую интерпретацию, следует отделить символы `<` и `-` друг от друга пробелом. На практике это вряд ли станет проблемой, поскольку не многие станут писать на Java `x<-y` без вставки пробелов или круглых скобок между операторами.

Смешанный идентификатор состоит из буквенно-цифрового идентификатора, за которым стоят знак подчеркивания и идентификатор оператора. Например, `unary_+`, использованный как имя метода, определяет унарный оператор `+`. А `myvar_=`, использованный как имя метода, определяет оператор

присваивания. Кроме того, смешанный идентификатор вида `myvar_=` генерируется компилятором Scala для поддержки свойств (более подробно этот вопрос рассматривается в главе 18).

Литеральный идентификатор представляет собой произвольную строку, заключенную в обратные кавычки (`...`). Примеры литеральных идентификаторов выглядят следующим образом:

```
`x` `<clinit>` `yield`
```

Замысел состоит в том, что между обратными кавычками можно поместить любую строку, которую среда выполнения воспримет в качестве идентификатора. В результате всегда будет получаться идентификатор Scala. Это сработает даже в том случае, если имя, заключенное в обратные кавычки, является в Scala зарезервированным словом. Обычно такие идентификаторы используются при обращении к статическому методу `yield` в Java-классе `Thread`. Вы не можете воспользоваться кодом `Thread.yield()`, поскольку в Scala `yield` является зарезервированным словом. Но имя метода все же можно использовать, если заключить его в обратные кавычки, например `Thread.`yield`()`.

6.11. Перегрузка методов

Вернемся к классу `Rational`. После внесения последних изменений появилась возможность применять операции сложения и умножения рациональных чисел в их естественном виде. Но нами все же упущена из виду смешанная арифметика. Например, вы не можете умножить рациональное число на целое, поскольку операнды у оператора `*` всегда должны быть `Rational`-объектами. Следовательно для рационального числа `r` вы не можете написать код `r * 2`. Вам нужно ввести `r * new Rational(2)`, а это имеет неприглядный вид.

Чтобы сделать класс Rational еще удобнее в использовании, добавим к нему новые методы, выполняющие смешанное сложение и умножение рациональных и целых чисел. А заодно введем методы вычитания и деления. Результат показан в листинге 6.5.

Листинг 6.5. Класс Rational с перегружаемыми методами

```
class Rational(n: Int, d: Int) {

  require(d != 0)

  private val g = gcd(n.abs, d.abs)
  val numer = n / g
  val denom = d / g
  def this(n: Int) = this(n, 1)

  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  def + (i: Int): Rational =
    new Rational(numer + i * denom, denom)

  def - (that: Rational): Rational =
    new Rational(
      numer * that.denom - that.numer * denom,
      denom * that.denom
    )
```

```
  def - (i: Int): Rational =
    new Rational(numer - i * denom, denom)

  def * (that: Rational): Rational =
      new Rational(numer * that.numer, denom *
that.denom)

  def * (i: Int): Rational =
    new Rational(numer * i, denom)

  def / (that: Rational): Rational =
      new Rational(numer * that.denom, denom *
that.numer)

  def / (i: Int): Rational =
    new Rational(numer, denom * i)

  override def toString = numer + "/" + denom

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
}
```

Теперь здесь две версии каждого арифметического метода: одна в качестве аргумента получает рациональное число, вторая — целое число. Иными словами, все эти методы называются перегружаемыми, поскольку каждое имя теперь применяется несколькими методами. Например, имя + используется и одним методом, который получает `Rational`-объект, и другим, который получает `Int`-объект. При вызове метода компилятор выбирает версию перегружаемого метода, которая в точности соответствует типу аргументов. Например, если аргумент `y` в вызове `x.+(y)` является `Rational`-объектом, компилятор выберет метод +,

получающий в качестве параметра Rational-объект. Но если аргумент является целым числом, компилятор выберет метод +, получающий в качестве параметра Int-объект. При запуске кода:

```
scala> val x = new Rational(2, 3)
x: Rational = 2/3

scala> x * x
res12: Rational = 4/9


scala> x * 2
res13: Rational = 4/3
```

станет понятно, что вызываемый метод * определяется всякий раз по типу его правого операнда.

ПРИМЕЧАНИЕ

В Scala процесс анализа при выборе перегружаемого метода очень похож на аналогичный процесс в Java. В любом случае выбранная перегружаемая версия является наиболее совпадающей со статическими типами аргументов. Иногда случается, что уникальной лучше всего соответствующей версии нет, и тогда компилятор выдаст ошибку, связанную с неоднозначной ссылкой, — ambiguous reference.

6.12. Подразумеваемое преобразование

Теперь, когда можно воспользоваться кодом r * 2, вы также можете захотеть поменять операнды местами, использовав код 2 * r. К сожалению, пока этот код работать не будет:

```
scala> 2 * r
<console>:10: error: overloaded method value * with

alternatives:
  (x: Double)Double <and>
  (x: Float)Float <and>
  (x: Long)Long <and>
  (x: Int)Int <and>
  (x: Char)Int <and>
  (x: Short)Int <and>
  (x: Byte)Int
cannot be applied to (Rational)
              2 * r

                ^
```

Проблема в том, что эквивалентом выражения `2 * r` является выражение `2.*(r)`, то есть вызов метода в отношении числа 2, которое является целым числом. Но в классе `Int` не содержится метода умножения, получающего в качестве аргумента `Rational`-объект, его там и не может быть, поскольку `Rational` не входит в число стандартных классов в библиотеке Scala.

Но в Scala есть другой способ решения этой проблемы: можно создать подразумеваемое преобразование, которое, когда понадобится, автоматически превратит целые числа в рациональные. Попробуйте добавить в интерпретатор следующую строку кода:

```
scala> implicit def intToRational(x: Int) = new Rational(x)
```

Она определяет метод преобразования из типа `Int` в тип `Rational`. Модификатор `implicit` перед определением метода сообщает компилятору о том, что в ряде ситуаций этот метод

следует применять автоматически. После определения преобразования можно еще раз попытаться выполнить код, который перед этим дал сбой:

```
scala> val r = new Rational(2,3)
r: Rational = 2/3

scala> 2 * r
res15: Rational = 4/3
```

Чтобы подразумеваемое преобразование заработало, оно должно находиться в области видимости. Если поместить определение метода с модификатором `implicit` внутрь класса `Rational`, он не попадет в область видимости интерпретатора. И в данном случае вам следует определить его непосредственно в интерпретаторе.

Из этого примера можно сделать вывод, что подразумеваемое преобразование является весьма эффективной технологией для придания библиотекам гибкости и повышения удобства их использования. Поскольку такие преобразования весьма серьезно воздействуют на код программы, в способах их применения можно легко наделать ошибок. Дополнительные сведения о подразумеваемых преобразованиях, а также способах их помещения в случае необходимости в область видимости будут даны в главе 21.

6.13. Предостережение

Создание методов с именами операторов и определение подразумеваемого преобразования, продемонстрированные в этой главе, призваны помочь в конструировании библиотек, для которых код клиента будет лаконичным и понятным. Scala предоставляет широкие возможности для разработки таких весьма доступных для использования библиотек. Но, пожалуйста, имейте в

виду, что, реализуя возможности, не следует забывать об ответственности.

При неумелом использовании и методы-операторы, и подразумеваемые преобразования могут сделать клиентский код таким, что его станет трудно читать и понимать. Поскольку выполнение компилятором подразумеваемого преобразования никак внешне не проявляется и не записывается в явном виде в исходный код, программистам-клиентам может быть невдомек, что именно оно и применяется в вашем коде. И хотя методы-операторы обычно делают клиентский код лаконичнее и читабельнее, таким он становится только для наиболее сведущих программистов-клиентов, способных запомнить и распознать значение каждого оператора.

При конструировании библиотек всегда нужно стремиться сделать клиентский код не просто лаконичным, но и легко читаемым и понимаемым. Читабельность может в значительной степени обусловливаться лаконичностью, которая способна заходить очень далеко. Конструируя библиотеки, позволяющие добиваться изысканной лаконичности и в то же время создавать понятный клиентский код, можно существенно поднять продуктивность работы использующих их программистов.

Резюме

В этой главе были рассмотрены многие аспекты классов Scala. Вы увидели способы добавления к классу параметров, определили несколько конструкторов, операторы и методы и настроили классы таким образом, чтобы их использование приобрело более естественный вид. Важнее всего, наверное, было показать вам, что определение и использование неизменяющихся объектов является вполне естественным способом программирования на Scala.

Хотя показанная здесь финальная версия класса `Rational` соответствует всем требованиям, обозначенным в начале главы, ее

можно еще усовершенствовать. Позже мы вернемся к этому примеру. В частности, в главе 30 будет рассмотрено переопределение равенств и хеш-кода, позволяющее `Rational`-объектам улучшить свое поведение при их сравнении с помощью оператора == или помещении в хеш-таблицы. В главе 21 поговорим о способах помещения определений подразумеваемых методов в объекты-спутники для класса `Rational`, которые упрощают для программистов-клиентов помещение в область видимости объектов типа `Rational`.

50 Метод require определен в обособленном Predef-объекте. Как упоминалось в разделе 4.4, элементы класса Predef автоматически импортируются в каждый исходный файл Scala.

51 Фактически Rational-объект можно сложить с самим собой, тогда ссылка будет на тот же самый объект, в отношении которого был вызван метод add. Но поскольку методу add можно передать любой Rational-объект, компилятор все же не позволит вам воспользоваться кодом that.n.

52 Параметрические поля, содержащие сокращения для написания аналогичного кода, будут рассматриваться в разделе 10.6.

53 Несмотря на то что n и d применяются в теле класса и учитывая, что они используются только внутри конструкторов, компилятор Scala не станет выделять под них поля. Таким образом, получив этот код, компилятор Scala создаст класс с двумя Int-полями, одно для numer, другое — для denom.

54 Этот стиль именования идентификаторов называется смешанным, или верблюжьим, поскольку у идентификаторов ИмеютсяГорбы, состоящие из символов в верхнем регистре.

55 В разделе 16.5 вы увидите, что иногда может возникнуть желание придать классу особый вид, как у case-класса, чье имя состоит только из символов оператора. Например, в API Scala имеется класс по имени ::, облегчающий поиск по шаблону в List-объектах.

56 Точнее, символ оператора принадлежит к математическим символам (Sm) или прочим символам (So) набора Unicode либо к 7-битным ASCII-символам, не являющимся буквами, цифрами, круглыми скобками, квадратными скобками, фигурными скобками, одинарными или двойными кавычками или знаками подчеркивания, точки, точки с запятой, запятой или обратных кавычек.

7. Встроенные структуры управления

В Scala имеется весьма незначительное количество встроенных управляющих структур. К ним относятся `if`, `while`, `for`, `try`, `match` и вызовы функций. Их в Scala немного, поскольку с момента создания этого языка в него были включены функциональные литералы. Вместо накопления в базовом синтаксисе одной за другой высокоуровневых систем управления Scala собирает их в библиотеках. Как именно это делается, будет показано в главе 9. А имеющиеся в Scala немногочисленные встроенные управляющие структуры будут рассмотрены в этой главе.

Следует учесть, что почти все управляющие структуры Scala приводят к какому-нибудь значению. Такой подход принят в функциональных языках, где программы рассматриваются в качестве вычислителей значений, стало быть, компоненты программы также должны вычислять значения. Можно рассматривать данное обстоятельство в качестве логического завершения тенденции, уже присутствующей в императивных языках. В них вызовы функций могут возвращать значение, даже когда наряду с этим будет работать обновление вызываемой функцией выходной переменной, переданной в качестве аргумента. Кроме этого, в императивных языках зачастую имеется тернарный оператор (такой как оператор `?:` в C, C++ и Java), который ведет себя точно так же, как `if`, но при этом выдает значение. Scala позаимствовал эту модель тернарного оператора, но назвал ее `if`. Иными словами, используемый в Scala оператор `if` может выдавать значение. Затем эта тенденция в Scala получила развитие: `for`, `try` и `match` тоже стали выдавать значения.

Программисты могут воспользоваться полученным в результате значением, чтобы упростить свой код, применяя те же приемы, что и для значений, возвращаемых функциями. Если бы этой особенности не существовало, программистам пришлось бы создавать временные переменные просто для хранения

результатов, вычисленных внутри управляющей структуры. Отказ от таких переменных немного упрощает код, а также избавляет от многих ошибок, возникающих, когда в одном ответвлении переменная создается, а в другом о ее создании забывают.

В целом основные управляющие структуры Scala в минимальном составе обеспечивают все, что нужно было взять из императивных языков. При этом они позволяют сделать код лаконичнее за счет неизменного наличия значений, получаемых в результате их применения. Чтобы показать все это в работе, далее более подробно рассмотрим основные управляющие структуры Scala.

7.1. Выражения if

Выражение `if` в Scala работает практически так же, как во многих других языках. Оно проверяет условие, а затем выполняет одну из двух ветвей кода в зависимости от того, вычисляется условие в `true` или нет. Простой пример, написанный в императивном стиле, выглядит следующим образом:

```
var filename = "default.txt"
if (!args.isEmpty)
  filename = args(0)
```

В этом коде объявляется переменная по имени `filename`, которая инициализируется значением по умолчанию. Затем в нем используется выражение `if`, чтобы проверить, предоставлены ли программе какие-либо аргументы. Если аргументы имеются, в переменную вносят изменения, чтобы в ней содержалось значение, указанное в списке аргументов. Если аргументы не предоставлялись, выражение оставляет значение переменной, установленное по умолчанию.

Этот код можно сделать гораздо выразительнее, поскольку, как упоминалось в шаге 3 главы 2, выражение `if` в Scala возвращает

значение. В листинге 7.1 показано, как можно выполнить те же самые действия, что и в предыдущем примере, не прибегая к использованию var-переменных:

Листинг 7.1. Особый стиль Scala, применяемый для условной инициализации

```
val filename =
  if (!args.isEmpty) args(0)
  else "default.txt"
```

На этот раз у if имеются два разветвления. Если массив args непустой, выбирается его начальный элемент args(0), в противном случае задается значение по умолчанию. Выражение if выдает результат в виде выбранного значения, которым инициализируется переменная filename. Этот код немного короче предыдущего, но гораздо существеннее то, что в нем задействуется val-, а не var-переменная. Использование val-переменной соответствует функциональному стилю и помогает вам примерно так же, как применение финальной (final) переменной в Java. Она сообщает читателям кода, что переменная никогда не изменится, избавляя их от необходимости просматривать весь код в области видимости переменной, чтобы понять, изменяется она где-нибудь или нет.

Второе преимущество использования вместо var-переменной val-переменной заключается в том, что она лучше поддерживает выводы, которые делаются посредством эквивалентных преобразований (equational reasoning). Введенная переменная равносильна вычисляющему выражению при условии, что у него нет побочных эффектов. Таким образом, всякий раз, собираясь написать имя переменной, вы можете вместо него написать выражение. Вместо println(filename), к примеру, просто напишите следующий код:

```
println(if (!args.isEmpty) args(0) else
"default.txt")
```

Выбор за вами. Можно написать все очень просто. Использование `val`-переменных помогает совершенно безопасно проводить подобную реструктуризацию кода по мере его развития.

Всегда изыскивайте возможности для применения `val`-переменных. Они смогут упростить не только чтение вашего кода, но и его реструктуризацию.

7.2. Циклы while

Используемые в Scala циклы `while` ведут себя точно так же, как и в других языках. В них имеются условие и тело, которое выполняется снова и снова, пока условие вычисляется в `true`. Пример показан в листинге 7.2.

Листинг 7.2. Вычисление наибольшего общего делителя с применением цикла while

```
def gcdLoop(x: Long, y: Long): Long = {
  var a = x
  var b = y
  while (a != 0) {
    val temp = a
    a = b % a
    b = temp
  }
  b
}
```

В Scala имеется также цикл `do-while`. Он работает точно так же, как и цикл `while`, но условие в нем проверяется после тела

цикла, а не до него. В листинге 7.3 показан сценарий на Scala, использующий цикл do-while для вывода на экран строк, считанных со стандартного ввода, до тех пор, пока не будет введена пустая строка.

Листинг 7.3. Чтение из стандартного ввода с применением цикла do-while

```
var line = ""
do {
  line = readLine()
  println("Read: " + line)
} while (line != "")
```

Конструкции while и do-while называются циклами, а не выражениями, поскольку в результате выполнения они не возвращают никакого содержательного значения. Типом результата является Unit. Получается так, что фактически существует только одно значение, имеющее тип Unit. Оно называется unit-значением и записывается как (). Существование () отличает имеющийся в Scala класс Unit от используемого в Java типа void. Попробуйте выполнить в интерпретаторе следующий код:

```
scala> def greet() = { println("hi") }
greet: ()Unit
scala> () == greet()
hi
res0: Boolean = true
```

Поскольку его телу не предшествует ни один знак равенства, greet определяется как процедура с типом результата Unit. Поэтому greet возвращает блоковое значение (). Это

подтверждается в следующей строке, где выполняется сравнение результата вызова `greet` с `unit`-значением `()`, выдающее `true`.

Еще одной конструкцией с типом результата в виде `unit`-значения, о которой здесь стоит упомянуть, является присваивание нового значения `var`-переменным. Например, если попробовать считать строки в Scala, воспользовавшись следующей идиомой цикла `while` из Java (а также из C и C++), возникнет проблема:

```
var line = ""
while ((line = readLine()) != "") // Этот код
работать не будет!
  println("Read: " + line)
```

При компиляции этого кода Scala выдаст предупреждение о том, что сравнение значений типа `Unit` и `String` с использованием `!=` всегда дает результат `true`. В то время как в Java присваивание выдает в качестве результата присваиваемое значение (в данном случае строку из стандартного ввода), в Scala оно всегда выдает `unit`-значение `()`. Таким образом, значением присваивания `line = readLine()` всегда будет `()`, но никак не `""`. В результате условие данного цикла `while` никогда не будет вычислено в `false` и из-за этого цикл никогда не завершится.

Поскольку в результате цикла `while` не возвращается никакого значения, его зачастую не включают в чисто функциональные языки. В таких языках имеются выражения, а не циклы. И тем не менее цикл `while` включен в Scala, поскольку иногда императивные решения бывает легче читать, особенно тем программистам, которые много работали с императивными языками программирования. Например, если нужно закодировать алгоритм, повторяющий процесс до тех пор, пока не изменятся какие-либо условия, циклом `while` можно выразить это напрямую, в то время как функциональную альтернативу наподобие использования рекурсии читателям кода распознавать оказывается сложнее.

Например, в листинге 7.4 показан альтернативный способ определения наибольшего общего знаменателя двух чисел[57]. При условии присвоения `x` и `y` в функции `gcd` таких же значений, как и функции `gcdLoop`, показанной в листинге 7.2, будет выдан точно такой же результат. Разница между этими двумя подходами состоит в том, что функция `gcdLoop` написана в императивном стиле с использованием `var`-переменных и цикла `while`, а функция `gcd` — в более функциональном стиле с применением рекурсии (`gcd` вызывает саму себя), для чего не нужны `var`-переменные.

Листинг 7.4. Вычисление наибольшего общего делителя с применением рекурсии

```
def gcd(x: Long, y: Long): Long =
  if (y == 0) x else gcd(y, x % y)
```

В целом мы рекомендуем относиться к циклам `while` в своем коде с оглядкой, как и к использованию в нем `var`-переменных. Фактически циклы `while` и `var`-переменные зачастую идут рука об руку. Поскольку циклы не дают результата в виде значения, для внесения в программу каких-либо изменений цикл `while` обычно будет нуждаться либо в обновлении `var`-переменных, либо в выполнении ввода-вывода. В этом можно убедиться, посмотрев на работу показанного ранее примера `gcdLoop`. По мере выполнения своей задачи этот цикл `while` обновляет значения `var`-переменных `a` и `b`. Поэтому мы советуем проявлять особую осмотрительность при использовании в коде циклов `while`. Если нет достаточных оснований для применения цикла `while` или `do-while`, попробуйте найти способ сделать то же самое без их участия.

7.3. Выражения for

Используемые в Scala выражения `for` являются для итераций чем-то наподобие швейцарского армейского ножа. Чтобы можно было реализовать широкий спектр итераций, они позволяют составлять комбинации из довольно простых ингредиентов различными способами. В результате можно решать простые задачи вроде поэлементного обхода последовательности целых чисел. Более сложные выражения могут выполнять обход элементов нескольких коллекций различных видов, фильтровать элементы на основе произвольных условий и создавать новые коллекции.

Обход элементов коллекций

Самое простое, что можно сделать с выражением `for`, — это выполнить обход всех элементов коллекции. Например, в листинге 7.5 показан код, который выводит имена всех файлов, содержащихся в текущем каталоге. Ввод-вывод выполняется с использованием API Java. Сначала в текущем каталоге «.» создается файл **java.io.File** и вызывается его метод `listFiles`. Этот метод возвращает массив `File`-объектов, по одному на каталог, и файл, содержащийся в текущем каталоге. Получившийся в результате массив сохраняется в переменной `filesHere`.

Листинг 7.5. Получение списка файлов в каталоге с применением выражения for

```
val filesHere = (new java.io.File(".")).listFiles

for (file <- filesHere)
  println(file)
```

С помощью синтаксиса `file <- filesHere`, называемого генератором, выполняется обход элементов массива `filesHere`.

При каждой итерации значением элемента инициализируется новая val-переменная по имени file. Компилятор приходит к выводу, что типом file является File, поскольку filesHere имеет тип Array[File]. Для каждой итерации выполняется тело выражения for, имеющее код println(file). Поскольку метод toString, определенный в классе File, выдает имя файла или каталога, будут выведены имена всех файлов и каталогов текущего каталога.

Синтаксис выражения for работает не только с массивами, но и с коллекциями любого типа[58]. Особым и весьма удобным случаем является применение типа Range, который был упомянут в табл. 5.4. Можно создавать Range-объекты, используя синтаксис вида 1 to 5, и выполнять их обход с применением for. Простой пример имеет следующий вид:

```
scala> for (i <- 1 to 4)
         println("Iteration " + i)
Iteration 1
Iteration 2
Iteration 3
Iteration 4
```

Если включать верхнюю границу диапазона в перечисляемые значения нежелательно, то вместо to используется until:

```
scala> for (i <- 1 until 4)
         println("Iteration " + i)
Iteration 1
Iteration 2
Iteration 3
```

Перебор целых чисел наподобие этого встречается в Scala довольно часто, но намного реже, чем в других языках, где этим свойством можно воспользоваться для перебора элементов

массива:

```
// В Scala такой код встречается довольно редко...
for (i <- 0 to filesHere.length - 1)
  println(filesHere(i))
```

В это выражение `for` введена переменная `i`, которой по очереди присваивается каждое целое число от `0` и до `filesHere.length - 1`, и для каждой установки `i` выполняется тело выражения. Для каждого значения `i` из массива `filesHere` извлекается и обрабатывается `i`-й элемент.

Причиной меньшей распространенности такого вида итераций в Scala является возможность непосредственного обхода элементов коллекции. При этом код становится короче и исключаются многие ошибки смещения на единицу, которые могут возникнуть при обходе элементов массива. С чего нужно начинать, с 0 или 1? Что следует прибавлять к завершающему индексу: –1, +1 или вообще ничего? На такие вопросы ответить несложно, но так же просто дать и неверный ответ. Безопаснее их вообще исключить.

Фильтрация

Иногда перебирать коллекцию целиком не нужно, а требуется отфильтровать ее в некий поднабор. В выражении `for` это можно сделать путем добавления фильтра в виде условия `if`, указанного в выражении `for` внутри круглых скобок. Например, код, показанный в листинге 7.6, выводит список только тех файлов текущего каталога, имена которых оканчиваются на `.scala`:

Листинг 7.6. Поиск файлов с расширением .scala с помощью for с фильтром

```
val filesHere = (new java.io.File(".")).listFiles
```

```
for (file <- filesHere if
file.getName.endsWith(".scala"))
  println(file)
```

Для достижения той же цели можно применить альтернативный вариант:

```
for (file <- filesHere)
  if (file.getName.endsWith(".scala"))
    println(file)
```

Этот код выдает на выходе то же самое, что и предыдущий, и выглядит, наверное, более привычно для программистов с опытом работы на императивных языках программирования. Но императивная форма — только вариант, поскольку данное выражение `for` выполняется для получения побочных эффектов, выражающихся в выводе данных, и выдает результат в виде `unit`-значения `()`. Чуть позже в этом разделе будет показано, что выражение `for` называется выражением, поскольку в результате его выполнения получается представляющий интерес результат, то есть коллекция, чей тип определяется компонентами `<-` выражения `for`.

Если потребуется, в выражение можно включить еще больше фильтров. В него просто нужно добавлять условия `if`. Например, для обеспечения безопасности код в листинге 7.7 выводит только файлы, исключая каталоги. Делается это путем добавления фильтра, проверяющего имеющийся у файла метод `isFile`.

Листинг 7.7. Использование в выражении for нескольких фильтров

```
for (
  file <- filesHere
  if file.isFile
  if file.getName.endsWith(".scala")
```

```
) println(file)
```

Вложенные итерации

Если добавить несколько операторов <-, будут получены вложенные циклы. Например, выражение for, показанное в листинге 7.8, имеет два вложенных цикла. Внешний цикл выполняет перебор элементов массива filesHere, а внутренний перебирает элементы fileLines(file) для каждого файла, имя которого оканчивается на .scala.

Листинг 7.8. Использование в выражении for нескольких генераторов

```
def fileLines(file: java.io.File) =
  scala.io.Source.fromFile(file).getLines().toList

def grep(pattern: String) =
  for (
    file <- filesHere
    if file.getName.endsWith(".scala");
    line <- fileLines(file)
    if line.trim.matches(pattern)
  ) println(file + ": " + line.trim)

grep(".*gcd.*")
```

При желании вокруг генераторов и фильтров вместо круглых скобок можно поставить фигурные. Одно из преимуществ использования фигурных скобок заключается в том, что они позволяют не ставить некоторые точки с запятой, которые нужны при использовании круглых скобок, поскольку, как объяснялось в разделе 4.2, компилятор Scala не будет подразумевать

использование точки с запятой внутри круглых скобок.

Привязки промежуточных переменных

Обратите внимание на повторение в предыдущем коде выражения `line.trim`. Это довольно сложное вычисление, поэтому его лучше выполнить один раз. Сделать это позволяет привязка результата к новой переменной с использованием знака равенства (=). Связанная переменная вводится и используется точно так же, как и `val`-переменная, но ключевое слово `val` не ставится. Пример показан в листинге 7.9.

Листинг 7.9. Промежуточное присваивание в выражении for

```
def grep(pattern: String) =
  for {
    file <- filesHere
    if file.getName.endsWith(".scala")
    line <- fileLines(file)
    trimmed = line.trim
    if trimmed.matches(pattern)
  } println(file + ": " + trimmed)

grep(".*gcd.*")
```

В листинге 7.9 переменная по имени `trimmed` вводится в ходе выполнения выражения `for`. Она инициализируется результатом вызова метода `line.trim`. Затем остальная часть кода выражения `for` использует новую переменную в двух местах: в выражении `if` и в методе `println`.

Создание новой коллекции

Хотя во всех рассмотренных до сих пор примерах вы работали со значениями, получаемыми при обходе элементов, после чего о них уже не вспоминали, у вас есть возможность создать значение, чтобы запомнить результат каждой итерации. Для этого нужно перед телом выражения `for` поставить ключевое слово `yield`. Рассмотрим, к примеру, функцию, идентифицирующую файлы с расширением `.scala` и сохраняющую их имена в массиве:

```
def scalaFiles =
  for {
    file <- filesHere
    if file.getName.endsWith(".scala")
  } yield file
```

При каждом выполнении тела выражения `for` создается одно значение, в данном случае это просто `file`. Когда выполнение выражения `for` завершится, результат будет включать все выданные значения, содержащиеся в единой коллекции. Тип получающейся коллекции задается на основе вида коллекции, обрабатываемой операторами итерации. В данном случае результат будет иметь тип `Array[File]`, поскольку `filesHere` является массивом, а выдаваемые выражением значения относятся к типу `File`.

Кстати, будьте внимательны при выборе места для ключевого слова `yield`. Синтаксис для выражения `for-yield` имеет следующий вид:

```
for условия yield тело
```

Ключевое слово `yield` ставится перед всем телом. Даже если тело является блоком, заключенным в фигурные скобки, `yield` нужно ставить перед первой фигурной скобкой, а не перед последним выражением блока. Не соблазняйтесь созданием кода, похожего на следующий:

```
for (file <- filesHere if
file.getName.endsWith(".scala")) {
  yield file // Синтаксическая ошибка!
}
```

Например, выражение `for`, показанное в листинге 7.10, сначала преобразует объект типа `Array[File]` по имени `filesHere`, в котором содержатся имена всех файлов, имеющихся в текущем каталоге, в объект, содержащий только имена файлов с расширением `.scala`. Для каждого из элементов создается объект типа `Iterator[String]`, являющийся результатом выполнения метода `fileLines`, чье определение показано в листинге 7.8. Класс `Iterator` предоставляет методы `next` и `hasNext`, позволяющие выполнять в коллекции обход элементов. Этот исходный итератор превращается в другой объект типа `Iterator[String]`, содержащий только те строки, обработанные методом `trim`, которые включают подстроку `for`. И наконец, для каждого из них выдается целочисленное значение длины. Результатом этого выражения `for` становится объект, имеющий тип `Array[Int]` и содержащий эти значения длины.

Листинг 7.10. Преобразование объекта типа Array[File] в объект типа Array[Int] с применением выражения for

```
val forLineLengths =
  for {
    file <- filesHere
    if file.getName.endsWith(".scala")
    line <- fileLines(file)
    trimmed = line.trim
    if trimmed.matches(".*for.*")
  } yield trimmed.length
```

Итак, основные свойства выражения `for`, применяемого в Scala, рассмотрены, но мы прошлись по ним слишком поверхностно. Более подробно о выражениях `for` поговорим в главе 23.

7.4. Обработка исключений с помощью выражений try

Исключения в Scala ведут себя практически так же, как во многих других языках. Вместо свойственного ему возвращения значения метод может прервать работу с выдачей исключения. Код, вызвавший метод, может либо перехватить и обработать это исключение, либо прекратить собственное выполнение, при этом передав исключение тому коду, который его вызвал. Исключение таким образом распространяется, отматывая назад стек вызова, до тех пор пока не встретится обрабатывающий его метод или вообще не останется ни одного метода.

Выдача исключений

Выдача исключений в Scala выглядит так же, как в Java. Создается объект исключения, который затем выдается с помощью ключевого слова `throw`:

```
throw new IllegalArgumentException
```

Как бы парадоксально это ни звучало, в Scala выдается исключение, у которого есть тип возвращаемого результата. Рассмотрим пример, где имеется тип возвращаемого значения:

```
val half =
  if (n % 2 == 0)
    n / 2
  else
    throw new RuntimeException("n must be even")
```

Здесь получается, что, если `n` является четным числом,

переменная `half` будет инициализирована половинным значением `n`. Если `n` не является четным числом, исключение будет выдано еще до того, как переменная `half` вообще будет инициализирована каким-либо значением. Поэтому выдаваемое исключение можно без малейших опасений рассматривать как некую разновидность значения. Любой контекст, пытающийся воспользоваться значением, возвращаемым из выдачи, никогда не сможет этого сделать, поэтому никакого вреда ожидать не приходится.

С технической точки зрения выданное исключение относится к типу `Nothing`. Выдачей исключения можно воспользоваться, даже если оно никогда и ни во что не будет вычислено. Подобные нюансы технических упражнений могут показаться несколько странными, но в случаях, подобных предыдущему примеру, они зачастую могут пригодиться. Одно ответвление от `if` вычисляет значение, а другое выдает исключение и вычисляет `Nothing`. Тогда типом всего выражения `if` является тип, вычисленный в той ветви, в которой проводилось вычисление. Тип `Nothing` дополнительно будет рассмотрен в разделе 11.3.

Перехват исключений

Перехват исключений выполняется с применением синтаксиса, показанного в листинге 7.11. Для выражений `catch` был выбран синтаксис с прицелом на совместимость с весьма важной частью — *поиском по шаблону*. Этот поиск является весьма эффективным свойством, которое вкратце рассматривается в данной главе, а более подробно — в главе 15.

Листинг 7.11. Применение в Scala конструкции try-catch

```
import java.io.FileReader
import java.io.FileNotFoundException
```

```
import java.io.IOException

try {
  val f = new FileReader("input.txt")
  // использование и закрытие файла
} catch {
   case ex: FileNotFoundException => // Обработка ошибки отсутствия файла
   case ex: IOException => // Обработка других ошибок ввода-вывода
}
```

Поведение выражения `try-catch` ничем не отличается от его поведения в других языках, использующих исключения. Если при выполнении тела выдается исключение, то по очереди предпринимается попытка выполнения каждого раздела `case`. Если в данном примере исключение имеет тип `FileNotFoundException`, будет выполнено первое условие. Если у него будет тип `IOException` — второе. Если исключение не относится ни к одному из этих типов, выражение `try-catch` прервет свое выполнение и исключение будет распространено далее.

ПРИМЕЧАНИЕ

Одно из отличий Scala от Java, которое довольно просто заметить, заключается в том, что язык Scala не требует от вас перехвата проверяемых исключений или их объявления в условии выдачи исключений. Если нужно, условие выдачи исключений можно объявить с применением аннотации @throws, но делать это необязательно. Дополнительную информацию о @throws можно найти в разделе 31.2.

Условие finally

Если нужно, чтобы какой-нибудь код выполнялся независимо от того, как именно было завершено выполнение выражения, можно воспользоваться условием `finally`, заключив в него этот код. Например, может возникнуть потребность в гарантированном закрытии открытого файла, даже если выход из метода произошел с выдачей исключения. Пример показан в листинге 7.1259.

Листинг 7.12. Применение в Scala условия try-finally

```
import java.io.FileReader

val file = new FileReader("input.txt")
try {
  // Использование файла
} finally {
  file.close() // Гарантированное закрытие файла
}
```

ПРИМЕЧАНИЕ

В листинге 7.12 показан характерный для языка способ гарантированного закрытия ресурса, не имеющего отношения к оперативной памяти, например файла, сокета или подключения к базе данных. Сначала вы получаете ресурс. Затем запускается на выполнение блок try, в котором этот ресурс используется. И наконец, вы закрываете ресурс в блоке finally. Этот способ характерен для Scala точно так же, как и для Java, но в качестве альтернативного варианта для достижения той же цели более лаконичным способом в Scala можно воспользоваться технологией под названием «шаблон временного пользования» (loan pattern).

Этот шаблон будет рассмотрен в разделе 9.4.

Выдача значения

Как и большинство других управляющих структур Scala, `try-catch-finally` выдает значение. Например, в листинге 7.13 показано, как можно попытаться разобрать URL-адрес, но при этом воспользоваться значением по умолчанию, если URL плохо сформирован. Результат получается при выполнении условия `try`, если не выдается исключение, или же при выполнении связанного с ним условия `catch`, если исключение выдается и перехватывается. Значение, вычисленное в условии `finally`, если таковое имеется, отбрасывается. Как правило, условия `finally` выполняют какую-либо подчистку, например закрытие файла. Обычно они не должны изменять значение, вычисленное в основном теле или в `catch`-условии, связанном с `try`.

Листинг 7.13. Условие catch, выдающее значение

```
import java.net.URL
import java.net.MalformedURLException

def urlFor(path: String) =
  try {
    new URL(path)
  } catch {
    case e: MalformedURLException =>
      new URL("http://www.scala-lang.org")
}
```

Если вы знакомы с Java, то стоит отметить, что поведение Scala отличается от поведения Java только потому, что используемая в Java конструкция `try-finally` не возвращает в результате

никакого значения. Как и в Java, если в условие `finally` включена в явном виде инструкция возвращения значения `return` или же в нем выдается исключение, то это возвращаемое значение или исключение будет перевешивать все, что ранее было выдано `try` или одним из его условий `catch`. Например, если взять вот такое несколько надуманное определение функции:

```
def f(): Int = try return 1 finally return 2
```

то при вызове `f()` будет получен результат 2. Для сравнения, если взять определение

```
def g(): Int = try 1 finally 2
```

то при вызове `g()` будет получен результат 1. Обе эти функции демонстрируют поведение, которое может удивить большинство программистов, поэтому все же лучше обойтись без значений, возвращаемых из условий `finally`. Условие `finally` предпочтительнее считать способом, гарантирующим выполнение какого-либо побочного эффекта, например закрытие открытого файла.

7.5. Выражения сопоставлений

Используемое в Scala выражение сопоставления позволяет сделать выбор из нескольких вариантов, как этот делается в других языках с помощью инструкции `switch`. В общем, выражение сопоставления позволяет использовать произвольные *шаблоны*, которые будут рассмотрены в главе 15. Общая форма может подождать. А пока нужно просто рассматривать использование сопоставления для выбора среди ряда альтернативных вариантов.

В качестве примера сценарий, показанный в листинге 7.14, считывает из списка аргументов название пищевого продукта и выводит пару к нему. Это выражение сопоставления анализирует

значение переменной `firstArg`, которое установлено на первый аргумент, извлеченный из списка аргументов. Если это строковое значение `salt` (соль), оно выводит `pepper` (перец), а если это `chips` (чипсы), то выводит `salsa` (острый соус) и т. д. Вариант по умолчанию указывается с помощью знака подчеркивания (_), который является подстановочным знаком, часто используемым в Scala в качестве заместителя для неизвестного значения.

Листинг 7.14. Выражение сопоставления с побочными эффектами

```
val firstArg = if (args.length > 0) args(0) else
""

firstArg match {
  case "salt" => println("pepper")
  case "chips" => println("salsa")
  case "eggs" => println("bacon")
  case _ => println("huh?")
}
```

От используемой в Java инструкции `switch` имеется несколько важных отличий. Одно из них заключается в том, что в `case`-инструкциях Scala наряду с прочим могут применяться любые разновидности констант, а не только константы целочисленного типа, константы-перечисления или строковые константы, как в `case`-инструкциях Java. В листинге 7.14 в качестве вариантов используются строки. Еще одно отличие заключается в том, что в конце каждого варианта нет инструкции `break`. Ее присутствие подразумевается, поэтому никакого перехода от одного варианта к другому не происходит. Теперь, без перехода к следующему варианту, код наиболее часто встречающегося варианта становится короче, а программисты избавляются от потенциального источника ошибок, возникавших ранее из-за нежелательного

перехода к следующему варианту, происходившего по их недосмотру.

Но, возможно, наиболее существенным отличием от `switch`-инструкции является то, что выражения сопоставления дают значение. В предыдущем примере в каждом варианте в выражении сопоставления на стандартное устройство выводится значение. Как показано в листинге 7.15, данный вариант будет работать так же хорошо и выдавать значение вместо его вывода на устройство. Значение, получаемое из этого выражения сопоставления, сохраняется в переменной `friend`. Кроме того что код становится короче (по крайней мере на несколько символов), теперь он выполняет две отдельные задачи: сначала выбирает продукт питания, а затем выводит его на устройство.

Листинг 7.15. Выражение сопоставления, выдающее значение

```
val firstArg = if (!args.isEmpty) args(0) else ""

val friend =
  firstArg match {
    case "salt" => "pepper"
    case "chips" => "salsa"
    case "eggs" => "bacon"
    case _ => "huh?"
  }

println(friend)
```

7.6. Жизнь без break и continue

Вероятно, вы заметили, что до сих пор не упоминалось ни о `break`, ни о `continue`. Из Scala эти инструкции исключены, поскольку они

плохо вписываются в функциональные литералы — свойство, которое описывается в следующей главе. Назначение `continue` в цикле `while` понятно, но что эта инструкция будет означать внутри функционального литерала? Так как в Scala поддерживаются оба стиля программирования, и императивный, и функциональный, то в данном случае из-за упрощения языка прослеживается небольшой перекос в сторону функционального программирования. Но волноваться не стоит. Существует множество способов программирования без `break` и `continue`, и если воспользоваться функциональными литералами, то варианты с ними зачастую могут быть короче исходного кода.

Простейший подход заключается в замене каждой инструкции `continue` условием `if`, а каждой инструкции `break` — булевой переменной. Эта переменная показывает, должен ли охватывающий цикл `while` продолжаться. Предположим, к примеру, что ведется поиск в списке аргументов строки, оканчивающейся на `.scala`, но не начинающейся с дефиса. В Java можно, отдавая предпочтение циклам `while`, а также инструкциям `break` и `continue`, написать следующий код:

```
int i = 0;    // Это код Java
boolean foundIt = false;
while (i < args.length) {
  if (args[i].startsWith("-")) {
    i = i + 1;
    continue;
  }
  if (args[i].endsWith(".scala")) {
    foundIt = true;
    break;
  }
  i = i + 1;
}
```

Для перекодировки этого фрагмента на Java непосредственно в код Scala вместо использования условия `if` с последующей инструкцией `continue` можно написать условие `if`, охватывающее всю оставшуюся часть цикла `while`. Чтобы избавиться от `break`, обычно добавляют булеву переменную, указывающую на необходимость продолжения, но в данном случае можно воспользоваться уже существующей переменной `foundIt`. При использовании этих двух приемов код приобретает вид, показанный в листинге 7.16.

Листинг 7.16. Выполнение цикла без break или continue

```
var i = 0
var foundIt = false

while (i < args.length && !foundIt) {
  if (!args(i).startsWith("-")) {
    if (args(i).endsWith(".scala"))
      foundIt = true
  }
  i = i + 1
}
```

Код Scala, показанный в листинге 7.16, очень похож на исходный код Java. Основные части остались на месте и располагаются в том же порядке. Используются две переменные с переприсваиваемыми значениями и цикл `while`. Внутри цикла выполняется проверка того, что `i` меньше `args.length`, а также проверка на наличие `"-"` и `".scala"`.

Если в коде листинга 7.16 нужно избавиться от `var`-переменных, попробуйте применить один из подходов, заключающийся в переписывании цикла в рекурсивную функцию. Можно, к примеру, определить функцию `searchFrom`,

получающую на входе целочисленное значение, выполняющую поиск с указанной им позиции, а затем возвращающую индекс желаемого аргумента. При использовании данного приема код приобретет вид, показанный в листинге 7.17.

Листинг 7.17. Рекурсивная альтернатива циклу с применением var-переменных

```
def searchFrom(i: Int): Int =
  if (i >= args.length) -1
  else if (args(i).startsWith("-")) searchFrom(i +
1)
  else if (args(i).endsWith(".scala")) i
  else searchFrom(i + 1)

val i = searchFrom(0)
```

В версии, показанной в листинге 7.17, в имени функции отображено ее предназначение, изложенное в понятной человеку форме, а в качестве замены цикла применяется рекурсия. Каждая инструкция `continue` заменена рекурсивным вызовом с использованием в качестве аргумента выражения `i + 1`, позволяющего эффективно переходить к следующему целочисленному значению. Многие программисты, привыкшие к рекурсиям, считают этот стиль программирования более наглядным.

ПРИМЕЧАНИЕ

В действительности компилятор Scala не будет выдавать для кода, показанного в листинге 7.17, рекурсивную функцию. Поскольку все рекурсивные вызовы находятся в позиции, завершающей вызов, компилятор создаст код, похожий на цикл while. Каждый

рекурсивный вызов будет реализован как возврат к началу функции. Оптимизация завершающих вызовов рассматривается в разделе 8.9.

Если после всего сказанного необходимость в использовании `break` не отпадет, вам поможет стандартная библиотека Scala. Для выхода из охватывающего блока кода, помеченного как `breakable` (допускающий выход), можно воспользоваться методом `break`, который предлагается в классе `Breaks` из пакета `scala.util.control`. Пример использования метода `break`, предоставляемого библиотекой, выглядит следующим образом:

```
import scala.util.control.Breaks._
import java.io._

val in = new BufferedReader(new
InputStreamReader(System.in))

breakable {
  while (true) {
    println("? ")
    if (in.readLine() == "") break
  }
}
```

Этот код будет многократно считывать непустые строки из устройства стандартного ввода. Как только пользователь введет пустую строку, поток управления выполнит выход из охватывающего блока `breakable`, а вместе с ним и из цикла `while`.

В классе `Breaks` метод `break` реализуется путем выдачи исключения, перехватываемого приложением, охватывающим метод в `breakable`-блоке. Поэтому вызову метода `break` не нужно

быть тем же самым методом, который запустил `breakable`-блок.

7.7. Область видимости переменных

Теперь, когда стало понятно, что такое встроенные управляющие структуры Scala, мы воспользуемся ими, чтобы объяснить, как в Scala работает область видимости.

> **Беглый просмотр для Java-программистов**
>
> Если вы программировали на Java, то увидите, что правила области видимости в Scala почти идентичны правилам, которые действуют в Java. Единственное различие между Java и Scala состоит в том, что в Scala позволяется определять переменные с одинаковыми именами во вложенных областях видимости. Поэтому при наличии навыков программирования на Java можно по крайней мере бегло просмотреть данный раздел.

У объявлений переменных в программах Scala имеется *область видимости*, определяющая, где конкретно может использоваться имя переменной. Самым распространенным примером определения области видимости является применение фигурных скобок, которые, как правило, вводят новую область видимости, и все, что определяется внутри этих скобок, теряет видимость после закрывающей скобки[60]. Рассмотрим в качестве иллюстрации функцию, показанную в листинге 7.18.

> **Листинг 7.18. Область видимости переменных при выводе на стандартное устройство таблицы умножения**

```
def printMultiTable() = {
```

```
  var i = 1
  // здесь в области видимости только переменная i
  while (i <= 10) {

    var j = 1
     // здесь в области видимости обе переменные,
как i, так и j
    while (j <= 10) {

      val prod = (i * j).toString
      // здесь в области видимости i, j и prod

      var k = prod.length
      // здесь в области видимости i, j, prod и k
      while (k < 4) {
        print(" ")
        k += 1
      }

      print(prod)
      j += 1
    }

    // i и j все еще в области видимости, а prod и
k — уже нет

    println()
    i += 1
  }
   // i все еще в области видимости, а j, prod и
k — уже нет
}
```

Функция printMultiTable, показанная в листинге 7.18, выводит таблицу умножения[61]. В первой инструкции этой функции вводится переменная i, которая инициализируется целым числом 1. Затем имя i можно использовать во всей остальной части функции.

Следующая инструкция в printMultiTable является циклом while:

```
while (i <= 10) {

  var j = 1
  …
}
```

Переменная i может использоваться здесь, поскольку она по-прежнему находится в области видимости. В первой инструкции внутри цикла while вводится еще одна переменная, которой дается имя j, и она также инициализируется значением 1. Поскольку переменная j была определена внутри открытого блока с фигурными скобками, который относится к циклу while, она может использоваться только внутри этого цикла while. При попытке что-либо сделать с j после закрывающей фигурной скобки цикла while после комментария, сообщающего, что j, prod и k уже вне области видимости, ваша программа не будет откомпилирована.

Все переменные, определенные в этом примере, — i, j, prod и k — являются локальными. Их локальность задается по отношению к функциям, в которых они определены. При каждом вызове функции используется новый набор локальных переменных.

После определения переменной задать новую переменную с таким же именем в той же области видимости уже нельзя. Например, следующий сценарий с двумя переменными по имени a в одной и той же области видимости откомпилирован не будет:

```
val a = 1
val a = 2 // Не откомпилируется
println(a)
```

В то же время во внешней области видимости вполне воможно определить переменную с точно таким же именем, что и во внутренней области видимости. Следующий сценарий будет откомпилирован и сможет быть запущен:

```
val a = 1;
{
  val a = 2 // Компилируется без проблем
  println(a)
}
println(a)
```

При выполнении данный сценарий выведет 2, а затем 1, поскольку переменная a, определенная внутри фигурных скобок, — это уже другая переменная, чья область видимости распространяется только до закрывающей фигурной скобки[62]. Следует отметить одно различие между Scala и Java, заключающееся в том, что Java не позволит создать во внутренней области видимости переменную, получающую такое же имя, которое имеется у переменной во внешней области видимости. В программе на Scala внутренняя переменная, как говорят, *затеняет* внешнюю переменную с точно таким же именем, поскольку внешняя переменная становится невидимой во внутренней области видимости.

Вы уже, наверное, замечали что-либо подобное эффекту затенения в интерпретаторе:

```
scala> val a = 1
a: Int = 1
```

```
scala> val a = 2
a: Int = 2

scala> println(a)
2
```

Там имена переменных можно использовать повторно как вам угодно. Кроме прочего, это позволяет передумать, если при первом определении переменной в интерпретаторе была допущена ошибка. Такая возможность появляется благодаря тому, что интерпретатор концептуально создает для каждой введенной вами инструкции новую вложенную область видимости. Таким образом, можно визуализировать ранее введенный код следующим образом:

```
val a = 1;
{
  val a = 2;
  {
    println(a)
  }
}
```

Этот код будет скомпилирован и запущен как сценарий на Scala и, подобно коду, набранному в интерпретаторе, выведет 2. Следует помнить, что такой код может сильно запутать читателей, поскольку имена переменных приобретают во вложенных областях видимости совершенно новый смысл. Зачастую вместо того, чтобы затенять внешнюю переменную, лучше выбрать для переменной новое узнаваемое имя.

7.8. Реорганизация кода, написанного в императивном стиле

Чтобы помочь вам вникнуть в функциональный стиль, в данном разделе будет реорганизован императивный подход к выводу таблицы умножения, показанной в листинге 7.18. Наша

функциональная альтернатива показана в листинге 7.19.

Листинг 7.19. Функциональный способ создания таблицы умножения

```
// Возвращение строчки в виде последовательности
def makeRowSeq(row: Int) =
  for (col <- 1 to 10) yield {
    val prod = (row * col).toString
    val padding = " " * (4 - prod.length)
    padding + prod
  }

// Возвращение строки в виде строкового значения
def makeRow(row: Int) = makeRowSeq(row).mkString

// Возвращение таблицы в виде строковых значений,
по одному значению
// на каждую строку
def multiTable() = {

   val tableSeq = // последовательность из строк
таблицы
    for (row <- 1 to 10)
    yield makeRow(row)

  tableSeq.mkString("\n")
}
```

Императивный стиль проявляется в листинге 7.18 двумя способами. Во-первых, побочным эффектом от вызова `printMultiTable` — выводом таблицы умножения на стандартное устройство. В листинге 7.19 функция реорганизована таким

образом, чтобы возвращать таблицу умножения в виде строкового значения. Поскольку функция больше не занимается выводом на стандартное устройство, она переименована в `multiTable`. Как уже упоминалось, одним из преимуществ функций, не имеющих побочных эффектов, является упрощение их блочного тестирования. Для тестирования `printMultiTable` понадобилось бы каким-то образом переопределять `print` и `println`, чтобы можно было проверить вывод на корректность. А протестировать `multiTable` можно гораздо проще — проверкой ее строкового результата.

Другими верными признаками императивного стиля в функции `printMultiTable` являются ее цикл `while` и `var`-переменные. В отличие от этого, в функции `multiTable` для выражений, вспомогательных функций и вызовов `mkString` используются `val`-переменные.

Чтобы облегчить чтение кода, мы реорганизовали две вспомогательные функции, `makeRow` и `makeRowSeq`. Функция `makeRowSeq` использует выражение `for`, чей генератор выполняет перебор номеров столбцов от 1 до 10. В теле этого `for`-выражения вычисляется произведение значения строки на значение столбца, определяется отступ, необходимый для произведения, выдается результат объединения строк отступа и произведения. Результатом выражения `for` будет последовательность (один из подклассов `scala.Seq`), содержащая выданные строки в качестве элементов. Другая вспомогательная функция, `makeRow`, просто вызывает метод `mkString` в отношении результата, возвращенного функцией `makeRowSeq`. Метод `mkString` объединяет строки, имеющиеся в последовательности, возвращая их в виде одной строки.

Метод `multiTable` сначала инициализирует `tableSeq` результатом выполнения выражения `for`, чей генератор перебирает числа от 1 до 10, чтобы для каждого вызова `makeRow` получалось строковое значение для данной строки таблицы.

Именно эта строка и выдается, поэтому результатом выполнения этого for-выражения будет последовательность строковых значений, представляющих строки таблицы. Остается только преобразовать последовательность строк в одну строку. Для выполнения этой задачи вызывается метод mkString, и, поскольку ему передается значение "\n", мы получаем символ конца строки, вставленный между всеми строками. Если передать строку, возвращенную multiTable, функции println, вы увидите, что выводится точно такая же таблица, что и при вызове функции printMultiTable:

```
1  2  3  4  5  6  7  8  9  10
2  4  6  8 10 12 14 16 18  20
3  6  9 12 15 18 21 24 27  30
4  8 12 16 20 24 28 32 36  40
5 10 15 20 25 30 35 40 45  50
6 12 18 24 30 36 42 48 54  60
7 14 21 28 35 42 49 56 63  70
8 16 24 32 40 48 56 64 72  80
9 18 27 36 45 54 63 72 81  90
10 20 30 40 50 60 70 80 90 100
```

Резюме

Перечень встроенных в Scala управляющих структур минимален, но эти структуры вполне справляются со своими задачами. Их работа похожа на действия их императивных эквивалентов, но, поскольку им свойственно выдавать значение, они поддерживают и функциональный стиль. Не менее важно и то, что они внимательны к тому, что пропускают, оставляя таким образом поле деятельности для одного из самых эффективных свойств Scala — функциональных литералов, которые рассматриваются в следующей главе.

57 Здесь в функции gcd используется точно такой же подход, который применялся в одноименной функции в листинге 6.3 для вычисления наибольших общих делителей для класса Rational. Основное отличие заключается в том, что вместо Int-значений gcd код работает с Long-значениями.

58 Точнее, выражение справа от символов <- в for-выражении может быть любого типа, имеющего определенные методы (в данном случае foreach) с соответствующими сигнатурами. Подробности того, как компилятор Scala обрабатывает выражения for, рассматриваются в главе 23.

59 Хотя инструкции case оператора catch всегда нужно заключать в круглые скобки, try и finally не требуют использования круглых скобок, если в них содержится только одно выражение. Например, можно написать: try t() catch { case e: Exception => ... } finally f().

60 Из этого правила есть несколько исключений, поскольку в Scala иногда можно вместо круглых добавить фигурные скобки. Одним из примеров подобного использования фигурных скобок является альтернативный синтаксис выражения for, рассмотренный в разделе 7.3.

61 Функция printMultiTable, показанная в листинге 7.18, написана в императивном стиле. В следующем разделе она будет приведена к функциональному стилю.

62 Кстати, в данном случае после первого определения нужно поставить точку с запятой, поскольку в противном случае действующий в Scala механизм, подразумевающий их использование, не сработает.

8. Функции и замыкания

Когда программы становятся больше, появляется необходимость их разбиения на небольшие части, которыми удобнее управлять. Чтобы разделить поток управления, в Scala реализуется подход, знакомый всем опытным программистам: разделение кода на функции. Фактически в Scala предлагается несколько способов определения функций, которых нет в Java. Кроме методов, представляющих собой функции, являющиеся частью объектов, есть также функции, вложенные в другие функции, функциональные литералы и функциональные значения. В этой главе вам предстоит познакомиться со всеми этими разновидностями функций, имеющимися в Scala.

8.1. Методы

Наиболее распространенным способом определения функций является включение их в состав объекта. Такая функция называется *методом*. В качестве примера в листинге 8.1 показаны два метода, которые вместе считывают данные из файла с заданным именем и выводят строки, длина которых превышает заданную. Перед каждой выведенной строкой указывается имя файла, в котором она появляется.

Листинг 8.1. LongLines с закрытым методом processLine

```
import scala.io.Source

object LongLines {
  def processFile(filename: String, width: Int) =
{
    val source = Source.fromFile(filename)
```

```
    for (line <- source.getLines())
      processLine(filename, width, line)
  }

  private def processLine(filename: String,
      width: Int, line: String) = {
    if (line.length > width)
      println(filename + ": " + line.trim)
  }
}
```

Метод `processFile` получает в качестве параметров имя файла и длину строки. Он создает `Source`-объект, а в отношении объекта `source` в генераторе выражения `for` вызывается метод `getLines`. Как упоминалось в шаге 12 главы 3, метод `getLines` возвращает итератор, который при каждой итерации выдает одну строку из файла, исключая при этом символ конца строки. Выражение `for` обрабатывает каждую из этих строк путем вызова вспомогательного метода `processLine`. Метод `processLine` получает три параметра: имя файла (`filename`), длину строки (`width`) и строку (`line`). Он проверяет длину строки на превышение заданной длины и при положительном результате выводит имя файла, двоеточие и строку.

Для использования `LongLines` из командной строки мы создадим приложение, ожидающее применения длины строки в качестве первого аргумента командной строки и интерпретирующее последующие аргументы в качестве имен файлов[63]:

```
object FindLongLines {
  def main(args: Array[String]) = {
    val width = args(0).toInt
    for (arg <- args.drop(1))
      LongLines.processFile(arg, width)
```

```
    }
  }
```

Для поиска в `LongLines.scala` строк, длина которых превышает 45 символов (там всего одна такая строка), это приложение можно использовать следующим образом:

```
$ scala FindLongLines 45 LongLines.scala
LongLines.scala: def processFile(filename: String,
width: Int) = {
```

До сих пор все это было очень похоже на то, что бы вы делали в любом объектно-ориентированном языке. Но концепция функции в Scala намного универсальнее применения простого метода. В этом языке программирования предлагаются другие способы выражения функций, рассматриваемые в следующих разделах.

8.2. Локальные функции

Конструкция метода `processFile` из предыдущего раздела продемонстрировала важность принципов разработки, присущих функциональному стилю программирования: программы должны быть разбиты на множество небольших функций с четко определенными задачами. Зачастую отдельно взятая функция имеет весьма небольшой размер. Преимуществом такого стиля является предоставление программисту множества строительных блоков, позволяющих составлять гибкие композиции при создании более сложных конструкций. Такие строительные блоки должны быть довольно простыми, чтобы с ними было легче разобраться по отдельности.

При таком подходе выявляется одна проблема: имена всех вспомогательных функций могут засорять пространство имен программы. В интерпретаторе это проявляется не так ярко, но как только функции будут упакованы в многократно используемые

классы и объекты, желательно скрыть вспомогательные функции от пользователей класса. Зачастую, будучи отдельно взятыми, такие функции не имеют особого смысла и возникает желание сохранять достаточную степень гибкости, чтобы можно было удалить вспомогательные функции, если позже класс будет переписан.

В Java основным инструментом для этого является закрытый метод. Как было продемонстрировано в листинге 8.1, точно такой же подход с использованием закрытого метода работает и в Scala, но в этом языке предлагается и еще один: можно определить функцию внутри другой функции. Как и локальные переменные, такие локальные функции видны только в пределах своего закрытого блока. Рассмотрим пример:

```
def processFile(filename: String, width: Int) = {

  def processLine(filename: String,
      width: Int, line: String) = {

    if (line.length > width)
      println(filename + ": " + line.trim)
  }

  val source = Source.fromFile(filename)
  for (line <- source.getLines()) {
    processLine(filename, width, line)
  }
}
```

Здесь реорганизована исходная версия `LongLines`, показанная в листинге 8.1: закрытый метод `processLine` был превращен в локальную функцию для `processFile`. Для этого был удален модификатор `private`, который мог быть применен только для

элементов класса (и это его единственное предназначение), а определение функции `processLine` было помещено внутрь определения функции `processFile`. В качестве локальной функция `processLine` находится в области видимости внутри функции `processFile`, за пределами которой она недоступна.

Но теперь, когда функция `processLine` определена внутри функции `processFile`, появилась возможность внесения еще одного усовершенствования. Вы заметили, что `filename` и `width` передаются вспомогательной функции, как и прежде? В этом нет никакой необходимости, поскольку локальные функции могут получать доступ к параметрам окружающей их функции. Как показано в листинге 8.2, можно просто воспользоваться параметрами внешней функции `processLine`.

Листинг 8.2. LongLines с локальной функцией processLine

```
import scala.io.Source

object LongLines {

  def processFile(filename: String, width: Int) =
{

    def processLine(line: String) = {
      if (line.length > width)
        println(filename + ": " + line.trim)
    }

    val source = Source.fromFile(filename)
    for (line <- source.getLines())
      processLine(line)
  }
```

```
}
```

Заметили, как упростился код? Такое использование параметров охватывающей функции является широко распространенным и весьма полезным примером предоставляемой языком Scala универсальной вложенности. Вложенность и области видимости применительно ко всем конструкциям языка Scala, включая функции, рассматриваются в разделе 7.7. Это довольно простой, но весьма эффективный принцип, особенно в языках, использующих функции первого класса.

8.3. Функции первого класса

В Scala имеются *функции первого класса*. Вы можете не только определить их и вызвать для выполнения, но и записать в виде безымянных *литералов*, после чего передать их в качестве *значений*. Понятие функциональных литералов было введено в главе 2, а их основной синтаксис показан на рис. 2.2.

Функциональный литерал компилируется в класс, который при создании экземпляра на этапе выполнения программы становится *функциональным значением*[64]. Таким образом, разница между функциональными литералами и значениями состоит в том, что функциональные литералы существуют в исходном коде, а функциональные значения существуют в виде объектов на этапе выполнения программы. Эта разница во многом похожа на разницу между классами (исходным кодом) и объектами (создаваемыми на этапе выполнения программы).

Простой функциональный литерал, прибавляющий к числу единицу, имеет следующий вид:

```
(x: Int) => x + 1
```

Сочетание символов `=>` указывает на то, что эта функция превращает то, что стоит слева от этого сочетания (любое

целочисленное значение x), в то, что находится справа от него (x + 1). Таким образом, данная функция отображает на любую целочисленную переменную x значение x + 1.

Функциональные значения являются объектами, следовательно, при желании их можно хранить в переменных. Они также являются функциями, следовательно, вы можете вызывать их, используя обычную систему записи вызова функций с применением круглых скобок. Вот как выглядят примеры обоих этих действий:

```
scala> var increase = (x: Int) => x + 1
increase: Int => Int = <function1>

scala> increase(10)
res0: Int = 11
```

Поскольку increase в данном примере является var-переменной, то позже ей можно присвоить другое значение:

```
scala> increase = (x: Int) => x + 9999
increase: Int => Int = <function1>

scala> increase(10)
res1: Int = 10009
```

Если нужно, чтобы в функциональном литерале использовалось более одной инструкции, следует заключить его тело в фигурные скобки и поместить каждую инструкцию на отдельной строке, сформировав блок. Как и в случае создания метода, когда вызывается функциональное значение, будут выполнены все инструкции и значением, возвращаемым из функции, станет значение, получаемое при вычислении последнего выражения:

```
scala> increase = (x: Int) => {
         println("We")
```

```
          println("are")
          println("here!")
          x + 1
        }
increase: Int => Int = <function1>

scala> increase(10)
We
are
here!
Res2: Int = 11
```

Итак, вы увидели все основные составляющие функциональных литералов и функциональных значений. Возможности их применения обеспечиваются многими библиотеками Scala. Например, методом `foreach`, доступным для всех коллекций[65]. Он получает функцию в качестве аргумента и вызывает ее в отношении каждого элемента своей коллекции. А вот как он может быть использован для вывода всех элементов списка:

```
scala> val someNumbers = List(-11, -10, -5, 0, 5,
10)
someNumbers: List[Int] = List(-11, -10, -5, 0, 5,
10)

scala> someNumbers.foreach((x: Int) => println(x))
-11
-10
-5
0
5
10
```

В другом примере используется также имеющийся у типов коллекций метод `filter`. Он выбирает те элементы коллекции, которые проходят выполняемую пользователем проверку с помощью функции. Например, для фильтрации может быть задействована функция `(x: Int) => x > 0`. Она отображает положительное целое число в `true`, а все остальные числа — в `false`. Метод `filter` можно использовать следующим образом:

```
scala> someNumbers.filter((x: Int) => x > 0)
res4: List[Int] = List(5, 10)
```

Более подробно о методах, подобных `foreach` и `filter`, поговорим позже. В главе 16 рассматривается их использование в классе `List`, а в главе 17 — применение с другими типами коллекций.

8.4. Краткие формы функциональных литералов

В Scala имеется несколько способов избавления от избыточной информации, позволяющих записывать функциональные литералы более кратко. Не упускайте такой возможности, поскольку это позволяет вам убрать из своего кода ненужный хлам.

Один из способов более лаконичного написания функциональных литералов заключается в отбрасывании типов параметров. При этом предыдущий пример с фильтром может быть написан следующим образом:

```
scala> someNumbers.filter((x) => x > 0)
res5: List[Int] = List(5, 10)
```

Компилятор Scala знает, что переменная `x` должна относиться к целым числам, поскольку видит, что вы сразу же применяете функцию для фильтрации списка целых чисел, на который ссылается объект `someNumbers`. Это называется *целевой*

типизацией, поскольку целевому использованию выражения — в данном случае в качестве аргумента для `someNumbers.filter()` — разрешено влиять на типизацию выражения — в данном случае на определение типа параметра `x`. Подробности целевой типизации нам сейчас не важны. Вы можете просто приступить к написанию функционального литерала без типа аргумента и, если компилятор не сможет в нем разобраться, добавить тип. Со временем вы начнете понимать, в каких ситуациях компилятор сможет, а в каких не сможет решить эту загадку.

Второй способ избавления от избыточных символов заключается в отказе от круглых скобок вокруг параметра, тип которого будет выведен автоматически. В предыдущем примере круглые скобки вокруг `x` совершенно излишни:

```
scala> someNumbers.filter(x => x > 0)
res6: List[Int] = List(5, 10)
```

8.5. Синтаксис заместителя

Чтобы сделать функциональный литерал еще короче, можно воспользоваться знаком подчеркивания в качестве заместителя для одного или нескольких параметров при условии, что каждый параметр появляется внутри функционального литерала только один раз. Например, `_ > 0` представляет собой очень краткую форму записи для функции, проверяющей, что значение больше нуля:

```
scala> someNumbers.filter(_ > 0)
res7: List[Int] = List(5, 10)
```

Знак подчеркивания можно рассматривать как бланк, который следует заполнить. Этот бланк будет заполнен аргументом функции при каждом ее вызове. Например, при условии, что

переменная someNumbers была здесь инициализирована значением List(-11, -10, -5, 0, 5, 10), метод filter заменит бланк в _ > 0 сначала значением -11, получив -11 > 0, затем значением -10, получив -10 > 0, затем -5, получив -5 > 0, и так далее до конца списка List. Таким образом, функциональный литерал _ > 0 является, как здесь показано, эквивалентом немного более пространного литерала x => x > 0:

```
scala> someNumbers.filter(x => x > 0)
res8: List[Int] = List(5, 10)
```

Иногда при использовании знаков подчеркивания в качестве заместителей параметров у компилятора может оказаться недостаточно информации для вывода неуказанных типов параметров. Предположим, к примеру, что вы сами написали _ + _:

```
scala> val f = _ + _
<console>:7: error: missing parameter type for
expanded
function ((x$1, x$2) => x$1.$plus(x$2))
       val f = _ + _
                   ^
```

В таких случаях нужно указать типы, используя двоеточие:

```
scala> val f = (_: Int) + (_: Int)
f: (Int, Int) => Int = <function2>

scala> f(5, 10)
res9: Int = 15
```

Следует заметить, что _ + _ расширяется в литерал для функции, получающей два параметра. Поэтому сокращенную форму можно использовать, только если каждый параметр

применяется в функциональном литерале не более одного раза. Несколько знаков подчеркивания означают наличие нескольких параметров, а не многократное использование одного и того же параметра. Первый знак подчеркивания представляет первый параметр, второй знак подчеркивания — второй параметр, третий знак подчеркивания — третий параметр и т. д.

8.6. Частично применяемые функции

В предыдущих примерах знаки подчеркивания ставились вместо отдельных параметров, но можно заменить знаком подчеркивания и весь список параметров. Например, вместо `println(_)` можно воспользоваться кодом `println _`. Рассмотрим код:

```
someNumbers.foreach(println _)
```

В Scala эта краткая форма интерпретируется так, будто вы воспользовались следующим кодом:

```
someNumbers.foreach(x => println(x))
```

Таким образом, знак подчеркивания в данном случае не является заместителем отдельного параметра. Он заменяет весь список параметров. Не забудьте поставить пробел между именем функции и знаком подчеркивания, в противном случае компилятор решит, что вы подразумеваете ссылку на другое обозначение, такое как обозначение метода по имени `println_`, которого, скорее всего, не существует.

Когда знак подчеркивания используется таким вот образом, создается частично применяемая функция. В Scala, когда при вызове функции в нее передаются любые необходимые аргументы, вы применяете ее к этим аргументам. Например, если имеется следующая функция:

```
scala> def sum(a: Int, b: Int, c: Int) = a + b + c
```

```
sum: (a: Int, b: Int, c: Int)Int
```

функцию sum можно применить к аргументам 1, 2 и 3 таким образом:

```
scala> sum(1, 2, 3)
res10: Int = 6
```

Частично применяемая функция является выражением, в котором не содержатся все аргументы, необходимые функции. Вместо этого есть лишь некоторые из них или вообще нет никаких необходимых аргументов. Например, для создания выражения частично применяемой функции, вызывающего sum, в котором вы не предоставляете ни одного из трех требуемых аргументов, вы помещаете знак подчеркивания после sum. Получившаяся функция может затем быть сохранена в переменной. Рассмотрим пример:

```
scala> val a = sum _
a: (Int, Int, Int) => Int = <function3>
```

При наличии данного кода компилятор Scala создает экземпляр функционального значения, получающего три целочисленных параметра, не указанных в выражении частично применяемой функции sum _, и присваивает ссылку на это новое функциональное значение переменной a. Когда к этому новому функциональному значению применяются три аргумента, оно развернется и вызовет sum, передав в нее те же три аргумента:

```
scala> a(1, 2, 3)
res11: Int = 6
```

Происходит следующее: переменная по имени a ссылается на объект функционального значения. Это функциональное значение является экземпляром класса, сгенерированного автоматически компилятором Scala из sum _ — выражения частично применяемой функции. Класс, сгенерированный компилятором, имеет метод

apply, получающий три аргумента[66]. Имеющийся в сгенерированном классе метод apply получает три аргумента, поскольку это и есть количество аргументов, отсутствующих в выражении sum _. Компилятор Scala транслирует выражение a(1, 2, 3) в вызов метода apply, принадлежащего объекту функционального значения, передавая ему три аргумента: 1, 2 и 3. Таким образом, a(1, 2, 3) является краткой формой следующего кода:

```
scala> a.apply(1, 2, 3)
res12: Int = 6
```

Метод apply, который определен в автоматически генерируемом компилятором Scala классе из выражения sum _, просто передает дальше эти три отсутствовавших параметра функции sum и возвращает результат. В данном случае метод apply вызывает sum(1, 2, 3) и возвращает то же, что и функция sum, то есть число 6.

Данный вид выражений, в которых знак подчеркивания используется для представления всего списка параметров, можно представить себе и по-другому — в качестве способа преобразования def в функциональное значение. Скажем, если имеется локальная функция, например sum(a: Int, b: Int, c: Int): Int, ее можно «завернуть» в функциональное значение, чей метод apply имеет точно такие же типы списка параметров и результата. Это функциональное значение, будучи примененным к неким аргументам, в свою очередь применяет sum для тех же самых аргументов и возвращает результат. Вы не можете присвоить переменной метод или вложенную функцию либо передать его или ее в качестве аргумента другой функции, однако все это можно сделать, если «завернуть» метод или вложенную функцию в функциональное значение путем помещения знака подчеркивания после имени метода или вложенной функции.

Теперь, несмотря на то что `sum _` действительно является частично применяемой функцией, может быть, вам не вполне понятно, почему она называется именно так. А все потому, что она применяется не ко всем своим аргументам. Что касается `sum _`, то функция не применяется ни к одному из своих аргументов. Но вы также можете выразить частично применяемую функцию, предоставив ей только некоторые из требуемых аргументов. Рассмотрим пример:

```
scala> val b = sum(1, _: Int, 3)
b: Int => Int = <function1>
```

В данном случае функции `sum` предоставлены первый и последний аргументы, а средний аргумент не указан. Поскольку пропущен только один аргумент, компилятор Scala сгенерирует новый функциональный класс, чей метод `apply` получает один аргумент. При вызове с этим одним аргументом метод `apply` сгенерированной функции вызывает функцию `sum`, передавая ей `1`, затем аргумент, переданный функции, и, наконец, `3`. Рассмотрим несколько примеров:

```
scala> b(2)
res13: Int = 6
```

В этом случае `b.apply` вызывает `sum(1, 2, 3)`:

```
scala> b(5)
res14: Int = 9
```

А в этом случае `b.apply` вызывает `sum(1, 5, 3)`.

Если функция требуется в конкретном месте кода, то при написании выражения частично применяемой функции, в котором не указан ни один параметр, например `println _` или `sum _`, его можно записать более кратко, не указывая знака подчеркивания. Например, вместо следующей записи вывода каждого числа,

имеющегося в объекте someNumbers (который определен в данном коде):

```
someNumbers.foreach(println _)
```

можно просто воспользоваться кодом

```
someNumbers.foreach(println)
```

Последняя форма допустима только в тех местах, где нужна функция (как вызов foreach в данном примере). Компилятор знает, что здесь требуется функция, поскольку foreach требует передачи функции в качестве аргумента. В тех ситуациях, когда функция не нужна, попытки применения данной формы записи приведут к ошибке компиляции. Рассмотрим пример:

```
scala> val c = sum
<console>:8: error: missing arguments for method sum;
follow this method with `_' if you want to treat it as a partially
applied function
       val c = sum
               ^

scala> val d = sum _
d: (Int, Int, Int) => Int = <function3>

scala> d(10, 20, 30)
res14: Int = 60
```

8.7. Замыкания

Все рассмотренные до сих пор в этой главе примеры

функциональных литералов ссылались только на передаваемые параметры. Например, в выражении `(x: Int) => x > 0` в теле функции `x > 0` использовалась только одна переменная `x`, которая определялась в качестве параметра функции. Но вы можете ссылаться на переменные, определенные и в других местах:

```
(x: Int) => x + more // на сколько больше?
```

Эта функция прибавляет значение переменной `more` к своему аргументу, но что такое `more`? С точки зрения данной функции `more` является свободной переменной, поскольку в самом функциональном литерале значение ей не присваивается. В отличие от нее, переменная `x` является связанной, так как в контексте функции у нее есть значение: она определена как единственный параметр функции, имеющий тип `Int`. Если попытаться воспользоваться этим функциональным литералом в чистом виде, без каких-либо определений в его области видимости, компилятор выразит свое недовольство:

```
scala> (x: Int) => x + more
<console>:8: error: not found: value more
              (x: Int) => x + more
                               ^
```

Зачем нужен конечный знак подчеркивания

Синтаксис, используемый в Scala для частично применяемых функций, высвечивает разницу при сопоставлении конструкций Scala и классических функциональных языков, таких как Haskell или ML. В этих языках использование частично применяемых функций в порядке вещей. Более того, в них применяется весьма строгая система статической типизации, которая обычно позволяет обнаружить каждую допущенную вами ошибку в

использовании частично применяемых функций. Scala имеет гораздо более близкое отношение к императивным языкам, таким как Java, где метод, не применяемый ко всем своим аргументам, считается ошибкой. Более того, объектно-ориентированные традиции порождения подтипов и универсального корневого типа, допускаемого некоторыми программами, в классических функциональных языках считались бы ошибочными.

К примеру, вы неправильно применили метод `drop(n: Int)` класса `List` для `tail()`, забыв, что нужно передать число для метода `drop`. Вы могли создать код `println(drop)`. Если бы Scala придерживался классической функциональной традиции, согласно которой частично применяемые функции могут находиться где угодно, этот код проходил бы проверку типа. Но, возможно, к вашему удивлению, инструкция `println` всегда бы выводила `<function>`! Чтобы выражение `drop` стало рассматриваться как функциональный объект, должно было что-то произойти. Поскольку `println` принимает объекты любого типа, код должен был успешно откомпилироваться, но при этом он выдал бы весьма неожиданный результат.

Чтобы избежать подобных ситуаций, в Scala обычно требуется указать специально пропущенные аргументы функции, даже если в качестве указателя используется просто знак подчеркивания (_). Scala позволяет обходиться без знака подчеркивания только тогда, когда ожидается функциональный тип.

В то же время тот же функциональный литерал будет нормально

работать, пока доступно что-нибудь с именем more:

```
scala> var more = 1
more: Int = 1

scala> val addMore = (x: Int) => x + more
addMore: Int => Int = <function1>

scala> addMore(10)
res16: Int = 11
```

Функциональное значение (объект), создаваемое на этапе выполнения программы из этого функционального литерала, называется *замыканием*. Такое название появилось из-за замыкания функционального литерала путем захвата привязок его свободных переменных. Функциональный литерал, не имеющий свободных переменных, например `(x: Int) => x + 1`, называется *замкнутым термом*, где *терм* представлен фрагментом исходного кода. Таким образом, функциональное значение, созданное на этапе выполнения программы из этого функционального литерала, строго говоря, не является замыканием, поскольку функциональный литерал `(x: Int) => x + 1` всегда замкнут уже по факту его написания. Но любой функциональный литерал со свободными переменными, например `(x: Int) => x + more`, является *открытым термом*. Поэтому каждое созданное на этапе выполнения программы из `(x: Int) => x + more` функциональное значение будет по определению требовать, чтобы привязка его свободной переменной `more` была захвачена. Получившееся функциональное значение, в котором может содержаться ссылка на захваченную переменную `more`, называется замыканием, поскольку функциональное значение является конечным продуктом замыкания открытого терма: `(x: Int) => x + more`.

Этот пример вызывает вопрос: что случится, если значение

more после создания замыкания изменится? В Scala можно ответить, что замыкание видит изменение, например:

```
scala> more = 9999
more: Int = 9999

scala> addMore(10)
res17: Int = 10009
```

На интуитивном уровне понятно, что замыкания в Scala перехватывают сами переменные, а не те значения, на которые они ссылаются[67]. Как показано в предыдущем примере, замыкание, созданное для (x: Int) => x + more, видит изменение more за пределами замыкания. То же самое справедливо и в обратном направлении. Изменения, вносимые в захваченную переменную, видимы за пределами замыкания. Рассмотрим пример:

```
scala> val someNumbers = List(-11, -10, -5, 0, 5,
10)
someNumbers: List[Int] = List(-11, -10, -5, 0, 5,
10)

scala> var sum = 0
sum: Int = 0

scala> someNumbers.foreach(sum += _)

scala> sum
res19: Int = -11
```

Здесь используется обходной способ сложения чисел в списке типа List. Переменная sum находится в области видимости, охватывающей функциональный литерал sum += _, который

прибавляет числа к sum. Несмотря на то что замыкание модифицирует sum на этапе выполнения программы, получающийся конечный результат -11 по-прежнему виден за пределами замыкания.

А что если замыкание обращается к некой переменной, у которой на этапе выполнения программы имеются несколько копий? Например, что если замыкание использует локальную переменную какой-нибудь функции и функция вызывается множество раз? Какой из экземпляров этой переменной будет задействован при каждом обращении?

С остальной частью языка согласуется только один ответ: задействуется тот экземпляр, который был активен на момент создания замыкания. Рассмотрим, к примеру, функцию, создающую и возвращающую замыкания прироста значения:

```
def makeIncreaser(more: Int) = (x: Int) => x + more
```

При каждом вызове эта функция будет создавать новое замыкание. Каждое замыкание будет обращаться к той переменной more, которая была активна при создании замыкания.

```
scala> val inc1 = makeIncreaser(1)
inc1: Int => Int = <function1>

scala> val inc9999 = makeIncreaser(9999)
inc9999: Int => Int = <function1>
```

При вызове makeIncreaser(1) создается и возвращается замыкание, захватывающее в качестве привязки к more значение 1. По аналогии с этим при вызове makeIncreaser(9999) возвращается замыкание, захватывающее для more значение 9999. Когда эти замыкания применяются к аргументам (в данном случае имеется только один передаваемый аргумент x), получаемый

результат зависит от того, как переменная `more` была определена в момент создания замыкания:

```
scala> inc1(10)
res20: Int = 11

scala> inc9999(10)
res21: Int = 10009
```

И неважно, что `more` в данном случае является параметром вызова метода, из которого уже произошло возвращение. В подобных случаях компилятор Scala осуществляет реорганизацию, позволяющую захваченным параметрам продолжать существовать в динамической памяти (куче), а не в стеке, и пережить таким образом метод, который их создал. Эта реорганизация происходит в автоматическом режиме, поэтому вам о ней не стоит беспокоиться. Захватывайте какую угодно переменную типа `val` или `var` или любой параметр.

8.8. Специальные формы вызова функций

Большинство встречающихся вам функций и вызовов функций будут такими, как те, которые вы уже видели в этой главе. У функции будет фиксированное число параметров, у вызова будет точно такое же количество аргументов, и аргументы будут указаны в точно таких же порядке и количестве, что и параметры.

Но поскольку вызовы функций в программировании на Scala весьма важны, для обеспечения некоторых специальных потребностей в язык были добавлены несколько форм определений и вызовов функций. В Scala поддерживаются повторяющиеся параметры, называемые аргументами и аргументами со значениями по умолчанию.

Повторяющиеся параметры

В Scala допускается указание на то, что последний параметр функции может повторяться. Это позволяет клиентам передавать функции список аргументов переменной длины. Чтобы обозначить повторяющийся параметр, поставьте после типа параметра знак звездочки, например:

```
scala> def echo(args: String*) =
         for (arg <- args) println(arg)

echo: (args: String*)Unit
```

Определенная таким образом функция `echo` может быть вызвана с нулем и большим количеством аргументов типа `String`:

```
scala> echo()

scala> echo("one")
one

scala> echo("hello", "world!")
hello
world!
```

Внутри функции типом повторяющегося параметра является `Array`, то есть массив элементов объявленного типа параметра. Таким образом, типом переменной `args` внутри функции `echo`, которая объявлена как тип `String*`, фактически является `Array[String]`. Несмотря на это, если у вас имеется массив подходящего типа, то при попытке передать его в качестве повторяющегося параметра будет получена ошибка компиляции:

```
scala> val arr = Array("What's", "up", "doc?")
arr: Array[String] = Array(What's, up, doc?)
```

```
scala> echo(arr)
<console>:10: error: type mismatch;
 found : Array[String]
 required: String
              echo(arr)
                   ^
```

Чтобы добиться успеха, нужно после аргумента в виде массива поставить двоеточие, знак подчеркивания и знак звездочки:

```
scala> echo(arr: _*)
What's
up
doc?
```

Эта система записи заставит компилятор передать в `echo` каждый элемент массива `arr` в виде самостоятельного аргумента, а не передавать его целиком в виде одного аргумента.

Именованные аргументы

При обычном вызове функции аргументы поочередно сопоставляются в указанном порядке с параметрами вызываемой функции:

```
scala> def speed(distance: Float, time: Float):
Float = distance / time

speed: (distance: Float, time: Float)Float

scala> speed(100, 10)
res27: Float = 10.0
```

В данном вызове `100` сопоставляется с `distance`, а `10` — с `time`. Сопоставление `100` и `10` производится в том же порядке, в котором

перечислены формальные параметры.

Именованные аргументы позволяют передавать аргументы функции в ином порядке. Синтаксис просто предусматривает указание перед каждым аргументом имени параметра и знака равенства. Например, следующий вызов функции `speed` эквивалентен вызову `speed(100,10)`:

```
scala> speed(distance = 100, time = 10)
res28: Float = 10.0
```

При вызове с именованными аргументами эти аргументы можно поменять местами, не изменяя их значения:

```
scala> speed(time = 10, distance = 100)
res29: Float = 10.0
```

Можно также смешивать позиционные и именованные аргументы. В этом случае сначала указываются позиционные аргументы. Именованные аргументы чаще всего используются в сочетании со значениями параметров по умолчанию.

Значения параметров, используемые по умолчанию

Scala позволяет указать для параметров функции значения по умолчанию. Аргумент для такого параметра может быть произвольно опущен из вызова функции, в таком случае соответствующий аргумент будет заполнен значением по умолчанию.

Пример показан в листинге 8.3. У функции `printTime` имеется один параметр, `out`, и у него есть значение по умолчанию `Console.out`.

Листинг 8.3. Параметр со значением по умолчанию

```
def printTime(out: java.io.PrintStream =
Console.out) =
            out.println("time = " +
System.currentTimeMillis())
```

Если функцию вызвать как `printTime()`, то есть без указания аргумента, используемого для `out`, для этого параметра будет установлено его значение по умолчанию `Console.out`. Можно также вызвать функцию с явно указанным выходным потоком. Например, путем вызова функции в виде `printTime(Console.err)` можно отправить регистрационную запись на стандартное устройство вывода сообщений об ошибках.

Параметры по умолчанию особенно полезны при использовании их в сочетании с именованными параметрами. В листинге 8.4 у функции `printTime2` есть два необязательных параметра. У параметра `out` имеется значение по умолчанию `Console.out`, а у параметра `divisor` — значение по умолчанию `1`.

Листинг 8.4. Функция с двумя параметрами, у которых имеются значения по умолчанию

```
def printTime2(out: java.io.PrintStream =
Console.out,
               divisor: Int = 1) =
            out.println("time = " +
System.currentTimeMillis()/divisor)
```

Чтобы оба параметра заполнялись их значениями по умолчанию, функцию `printTime2` можно вызывать как `printTime2()`. Но при использовании именованных аргументов может быть указан любой из параметров, а для другого останется значение по умолчанию. При необходимости указания выходного потока вызов выглядит следующим образом:

```
printTime2(out = Console.err)
```

Для указания делителя времени используется такой вызов:

```
printTime2(divisor = 1000)
```

8.9. Концевая рекурсия

В разделе 7.2 упоминалось, что для преобразования цикла `while`, обновляющего значение `var`-переменных, в код более функционального стиля, использующий только `val`-переменные, временами может понадобиться использование рекурсии. Рассмотрим пример рекурсивной функции, вычисляющей приблизительное значение путем повторяющегося уточнения приблизительного расчета до получения приемлемого результата:

```
def approximate(guess: Double): Double =
  if (isGoodEnough(guess)) guess
  else approximate(improve(guess))
```

С соответствующими реализациями `isGoodEnough` и `improve` подобные функции часто используются при решении поисковых задач. Если нужно, чтобы функция `approximate` выполнялась быстрее, может возникнуть желание написать ее с циклом `while`:

```
def approximateLoop(initialGuess: Double): Double
= {
  var guess = initialGuess
  while (!isGoodEnough(guess))
    guess = improve(guess)
  guess
}
```

Какая из двух версий приближения предпочтительнее? Если требуется лаконичность и нужно избавиться от использования

var-переменных, выиграет первый, функциональный вариант. Но, может быть, эффективнее окажется императивный подход? На самом деле, если измерить время выполнения, окажется, что они практически одинаковы!

Этот результат может стать неожиданным, поскольку рекурсивный вызов выглядит намного более затратным, чем простой переход из конца цикла в его начало. Но в показанном ранее вычислении приблизительного значения компилятор Scala может применить весьма существенную оптимизацию. Обратите внимание на то, что в вычислении тела функции `approximate` рекурсивный вызов стоит в самом конце. Функции наподобие `approximate`, которые в качестве последнего действия вызывают самих себя, называются *функциями с концевой рекурсией*. Компилятор Scala обнаруживает концевую рекурсию и заменяет ее переходом к началу функции после обновления параметров функции новыми значениями.

Из этого можно сделать вывод, что избегать использования рекурсивных алгоритмов для решения ваших задач не стоит. Зачастую рекурсивное решение выглядит более элегантно и лаконично, чем решение на основе цикла. Если в решении используется концевая рекурсия, то расплачиваться за него издержками производительности на этапе выполнения программы не придется.

Трассировка функций с концевой рекурсией

Для каждого вызова функции с концевой рекурсией новый стековый фрейм создаваться не будет, все вызовы станут выполняться с использованием одного и того же фрейма. Это обстоятельство может вызвать удивление у программиста, исследующего трассировку стека программы, давшей сбой. Рассмотрим, к примеру, несколько самовызовов этой функции с последующей выдачей исключения:

```
def boom(x: Int): Int =
```

```
  if (x == 0) throw new Exception("boom!")
  else boom(x - 1) + 1
```

Эта функция не относится к функциям с концевой рекурсией, поскольку после рекурсивного вызова она выполняет операцию инкремента. При ее запуске будет получен вполне ожидаемый результат:

```
scala> boom(3)
java.lang.Exception: boom!
      At .boom(<console>:5)
      at .boom(<console>:6)
      at .boom(<console>:6)
      at .boom(<console>:6)
      at .<init>(<console>:6)
…
```

Оптимизация концевого вызова

Код, скомпилированный для `approximate`, по сути, такой же, как и код, скомпилированный для `approximateLoop`. Обе функции компилируются в одни и те же 13 инструкций байт-кода Java. Если просмотреть байт-коды, сгенерированные компилятором Scala для метода с концевой рекурсией `approximate`, можно увидеть, что, хотя и `isGoodEnough`, и `improve` вызываются в теле метода, `approximate` там не вызывается. При оптимизации компилятор Scala убирает рекурсивный вызов:

```
public double approximate(double);
Code:
0: aload_0
1: astore_3
2: aload_0
```

```
3: dload_1
4: invokevirtual #24; // Метод isGoodEnough:(D)Z
7: ifeq 12
10: dload_1
11: dreturn
12: aload_0
13: dload_1
14: invokevirtual #27; // Метод improve:(D)D 17:
dstore_1
18: goto 2
```

Если теперь внести изменения в `boom`, чтобы в функции появилась концевая рекурсия:

```
def bang(x: Int): Int =
  if (x == 0) throw new Exception("bang!")
  else bang(x - 1)
```

вы получите следующий результат:

```
scala> bang(5)
java.lang.Exception: bang!
      At .bang(<console>:5)
      at .<init>(<console>:6) ...
```

На этот раз для `bang` виден только один стековый фрейм. Можно подумать, что `bang` дает сбой перед своим собственным вызовом, но это не так. Если вы считаете, что при просмотре трассировки стека оптимизации концевых вызовов могут вас запутать, их можно выключить, указав для оболочки `scala` или компилятора `scalac` следующий ключ:

```
-g:notailcalls
```

При указании этого ключа вы получите удлиненную

трассировку стека:

```
scala> bang(5)
java.lang.Exception: bang!
      At .bang(<console>:5)
      at .bang(<console>:5)
      at .bang(<console>:5)
      at .bang(<console>:5)
      at .bang(<console>:5)
      at .bang(<console>:5)
      at .<init>(<console>:6) …
```

Ограничения концевой рекурсии

Использование концевой рекурсии в Scala имеет строгие ограничения, поскольку набор инструкций виртуальной машины Java, JVM, существенно затрудняет реализацию более сложных форм концевых рекурсий. Оптимизация в Scala касается только непосредственных рекурсивных вызовов назад к той же самой функции, из которой выполняется вызов. Если рекурсия не является непосредственной, как в следующем примере, где используются две взаимно рекурсивные функции, ее оптимизация невозможна:

```
def isEven(x: Int): Boolean =
  if (x == 0) true else isOdd(x - 1)
def isOdd(x: Int): Boolean =
  if (x == 0) false else isEven(x - 1)
```

Получить оптимизацию концевого вызова невозможно и в том случае, если завершающий вызов делается в отношении функционального значения. Рассмотрим, к примеру, следующий рекурсивный код:

```
val funValue = nestedFun _
```

```
def nestedFun(x: Int) : Unit = {
  if (x != 0) { println(x); funValue(x - 1) }
}
```

Переменная `funValue` ссылается на функциональное значение, которое, по сути, заключает в себе вызов функции `nestedFun`. При применении функционального значения к аргументу все изменяется и `nestedFun` применяется к тому же самому аргументу, возвращая результат. Поэтому вы можете понадеяться на то, что компилятор Scala выполнит оптимизацию концевого вызова, но в данном случае этого не произойдет. Оптимизация концевых вызовов ограничивается ситуациями, когда метод или вложенная функция вызывают сами себя непосредственно в качестве своей последней операции, без прохода по функциональному значению или через какого-то другого посредника. (Если вы еще не усвоили, что такое концевая рекурсия, перечитайте раздел 8.9.)

Резюме

В этой главе был представлен довольно подробный обзор использования функций в языке Scala. Кроме методов, в Scala имеются локальные функции, функциональные литералы и функциональные значения. В дополнение к обычным вызовам функций в Scala используются частично применяемые функции и функции с повторяющимися параметрами. При благоприятной возможности вызовы функций реализуются в виде оптимизированных концевых вызовов, благодаря чему многие привлекательные рекурсивные функции выполняются практически так же быстро, как и оптимизированные вручную версии, использующие циклы `while`. В следующей главе на основе этих основных положений будет показано, как имеющаяся в Scala расширенная поддержка функций помогает абстрагировать процессы управления.

63 В примерах приложений, которые даются в книге, проверка аргументов командной строки чаще всего проводиться не будет как из соображений экономии лесных угодий, так и в силу стремления избавиться от повторяющегося кода, который может заслонить собой важный код примеров. Издержки такого решения будут выражаться в том, что вместо выдачи информативного сообщения об ошибке в случае введения неверных данных наши примеры приложений будут выдавать исключение.

64 Каждое функциональное значение является экземпляром какого-нибудь класса, представляющего собой расширение одного из нескольких FunctionN-трейтов в пакете scala, например Function0 для функций без параметров, Function1 для функций с одним параметром и т. д. В каждом трейте FunctionN имеется метод apply, используемый для вызова функции.

65 Метод foreach определен в трейте Traversable, который является родительским трейтом для List, Set, Array и Map. Подробности — в главе 17.

66 Сгенерированный класс является расширением трейта Function3, в котором объявлен метод apply, предусматривающий использование трех аргументов.

67 В отличие от этого, внутренние классы Java вообще не позволяют обращаться к модифицируемым переменным в окружающей области видимости, поэтому нет никакой разницы между захватом переменной и захватом того значения, которое в ней находится в данный момент.

9. Управляющие абстракции

В главе 7 было отмечено, что встроенных управляющих абстракций в Scala не так уж много, поскольку этот язык позволяет вам создавать собственные управляющие абстракции. В предыдущей главе мы рассмотрели функциональные значения. В этой главе будут показаны способы применения функциональных значений для создания новых управляющих абстракций. Попутно рассмотрим карринг параметров и параметры до востребования.

9.1. Сокращение повторяемости кода

Функции делятся на общие части, одинаковые для всех вызовов функции, и особые части, которые могут варьироваться от одного вызова функции к другому. Общие части находятся в теле функции, а особые части должны предоставляться через аргументы. Когда в качестве аргумента используется функциональное значение, особая часть алгоритма сама по себе является еще одним алгоритмом! При каждом вызове такой функции ей можно передавать в качестве аргумента другое функциональное значение, и функция по своему выбору вызывает переданное функциональное значение. Такие *функции высшего порядка*, то есть функции, получающие другие функции в качестве параметров, обеспечивают вам дополнительные возможности по сокращению и упрощению кода.

Одним из преимуществ функций высшего порядка является предоставление вам возможности создания управляющих абстракций, позволяющих избавиться от повторяющихся фрагментов кода. Предположим, к примеру, что вы создаете обозреватель файлов и должны разработать API, разрешающий пользователям вести поиск файлов, соответствующих какому-либо критерию. Сначала вы добавляете средство поиска тех файлов, чьи имена заканчиваются конкретной строкой. Это даст пользователям

возможность найти, к примеру, все файлы с расширением **.scala**. Такой API можно создать путем определения открытого метода `filesEnding` внутри следующего синглтон-объекта:

```
object FileMatcher {
        private def filesHere = (new
java.io.File(".")).listFiles
  def filesEnding(query: String) =
             for (file <- filesHere; if
file.getName.endsWith(query))
      yield file
}
```

Метод `filesEnding` получает список всех файлов, находящихся в текущем каталоге, используя закрытый вспомогательный метод `filesHere`, затем фильтрует этот список на основе завершения имени каждого файла тем содержимым, которое указано в пользовательском запросе. Поскольку `filesHere` является закрытым методом, метод `filesEnding` — единственный доступный метод, определенный в `FileMatcher`, то есть в API, который вы предлагаете своим пользователям.

Пока все идет неплохо — повторяющегося кода нет. Но чуть позже вы принимаете решение о допустимости ведения поиска по любой части имени файла. Такой поиск пригодится, когда пользователи не смогут вспомнить, как именно они назвали файл, **phb-important.doc**, **stupid-phb-report.doc**, **may2003salesdoc.phb** или совершенно иначе, и единственное, в чем они уверены, что где-то в имени фигурирует `phb`. Работа возобновляется, и к `FileMatcher` API добавляется соответствующая функция:

```
def filesContaining(query: String) =
         for (file <- filesHere; if
file.getName.contains(query))
    yield file
```

Эта функция работает точно так же, как и `filesEnding`. Она ищет текущие файлы с помощью `filesHere`, проверяет имя и возвращает файл, если его имя соответствует критерию поиска. Единственным отличием является использование этой функцией метода `contains` вместо метода `endsWith`. Проходит несколько месяцев, и программа набирает популярность. Со временем вы уступаете просьбам некоторых активных пользователей, желающих вести поиск с помощью регулярных выражений. У этих нерадивых пользователей образовались огромные каталоги с тысячами файлов, и им хочется получить возможность поиска всех PDF-файлов, у которых где-нибудь в названии имеется сочетание `oopsla`. Чтобы позволить им сделать это, вы создаете следующую функцию:

```
def filesRegex(query: String) =
        for (file <- filesHere; if
file.getName.matches(query))
    yield file
```

Опытные программисты могут обратить внимание на все допущенные повторения и удивиться тому, почему они не были сведены в общую вспомогательную функцию. Но если подходить к решению этой задачи в лоб, ничего не получится. Можно было бы придумать следующее:

```
def filesMatching(query: String, метод) =
        for (file <- filesHere; if
file.getName.метод(query))
    yield file
```

В некоторых динамичных языках такой подход сработал бы, но в Scala не разрешается объединять подобный код на этапе выполнения. Так что же делать?

Ответ дают функциональные значения. Раздавать имя метода в качестве значения нельзя, но точно такой же эффект можно

получить путем раздачи функционального значения, вызывающего для вас этот метод. В этом случае к методу, чьей единственной задачей будет проверка соответствия имени файла запросу, добавляется параметр `matcher`:

```
def filesMatching(query: String,
    matcher: (String, String) => Boolean) = {
  for (file <- filesHere; if matcher(file.getName,
query))
    yield file
}
```

В этой версии метода условие `if` теперь использует `matcher` для проверки соответствия имени файла запросу. Что именно проверяется, зависит от того, что указано в качестве `matcher`. А теперь посмотрите на тип самого параметра `matcher`. Это функция, поэтому в типе имеется обозначение `=>`. Функция получает два строковых аргумента, имя файла и запрос, и возвращает булево значение, следовательно, типом этой функции является `(String, String) => Boolean`.

Располагая новым вспомогательным методом по имени `filesMatching`, можно упростить три поисковых метода, заставив их вызывать вспомогательный метод, передавая в него соответствующую функцию:

```
def filesEnding(query: String) =
  filesMatching(query, _.endsWith(_))

def filesContaining(query: String) =
  filesMatching(query, _.contains(_))

def filesRegex(query: String) =
  filesMatching(query, _.matches(_))
```

Функциональные литералы, показанные в этом примере, используют синтаксис заместителя, рассмотренный в предыдущей главе, который может быть вам еще не совсем привычен. Поэтому поясним, как применяются заместители: функциональный литерал `_.endsWith(_)`, используемый в методе `filesEnding`, означает то же самое, что и следующий код:

```
(fileName: String, query: String) =>
fileName.endsWith(query)
```

Поскольку `filesMatching` получает функцию, требующую два `String`-аргумента, типы аргументов указывать не нужно — можно просто воспользоваться кодом `(ilename, query) => ilename.endsWith(query)`. А так как в теле функции каждый из параметров используется только один раз (то есть первый параметр, `ilename`, применяется в теле первым, второй параметр, `query`, — вторым), можно воспользоваться синтаксисом заместителей — `_.endsWith(_)`. Первый знак подчеркивания станет заместителем для первого параметра — имени файла, второй — заместителем для второго параметра — строки запроса.

Этот код уже упрощен, но он может стать еще короче. Заметьте, что аргумент `query` передается `filesMatching`, но эта функция ничего с ним не делает, за исключением того, что передает его назад переданной функции `matcher`. Такая передача туда-сюда необязательна, поскольку вызывающий код с самого начала знает о `query`! Это позволяет удалить параметр `query` как из `filesMatching`, так и из `matcher`, упростив код до состояния, показанного в листинге 9.1.

Листинг 9.1. Использование замыканий для сокращения повторяющихся фрагментов кода

```
object FileMatcher {
```

```
        private def filesHere = (new
java.io.File(".")).listFiles

   private def filesMatching(matcher: String =>
Boolean) =
             for (file <- filesHere; if
matcher(file.getName))
      yield file

  def filesEnding(query: String) =
    filesMatching(_.endsWith(query))

  def filesContaining(query: String) =
    filesMatching(_.contains(query))

  def filesRegex(query: String) =
    filesMatching(_.matches(query))
}
```

В этом примере продемонстрирован способ, позволяющий функциям первого класса помочь вам избавиться от повторяющихся фрагментов кода, что без них было бы очень трудно сделать. В Java, к примеру, можно создать интерфейс, содержащий метод, получающий одно `String`-значение и возвращающий значение типа `Boolean`, а затем добавить и передать функции `filesMatching` экземпляры безымянного внутреннего класса, реализующие этот интерфейс. Хотя такой подход позволит избавиться от повторяющихся фрагментов кода, чего, собственно, вы и добивались, но в то же время приведет к добавлению чуть ли не большего количества нового кода. Стало быть, цена вопроса сведет на нет все преимущества и лучше будет, наверное, смириться с повторяемостью кода.

Кроме того, данный пример показывает, как замыкания

способны помочь сократить повторяемость кода. Функциональные литералы, использованные в предыдущем примере, такие как `_.endsWith(_)` и `_.contains(_)`, на этапе выполнения программы становятся экземплярами функциональных значений, не являющимися замыканиями, поскольку они не захватывают никаких свободных переменных. К примеру, обе переменные, использованные в выражении `_.endsWith(_)`, представлены в виде знаков подчеркивания, следовательно, они берутся из аргументов функции. Таким образом, в `_.endsWith(_)` применяются не свободные, а две связанные переменные. В отличие от этого, в функциональном литерале `_.endsWith(query)`, использованном в самом последнем примере, содержатся одна связанная переменная, а именно аргумент, представленный знаком подчеркивания, и одна свободная переменная по имени `query`. Возможность убрать параметр `query` из `filesMatching` в этом примере, упростив тем самым код еще больше, появилась у вас только потому, что в Scala поддерживаются замыкания.

9.2. Упрощение клиентского кода

В предыдущем примере было показано, что применение функций высшего порядка способствует сокращению повторяемости кода, что и делалось при реализации API. Еще одним важным способом использования функций высшего порядка является помещение их в сам API с целью повышения лаконичности клиентского кода. Хорошим примером могут послужить методы организации циклов специального назначения, принадлежащие имеющимся в Scala типам коллекций[68]. Многие из них перечислены в табл. 3.1, но сейчас, чтобы понять, почему эти методы настолько полезны, внимательно рассмотрите только один пример.

Разберем `exists` — метод, определяющий факт наличия переданного значения в коллекции. Разумеется, искать элемент

можно, инициализировав var-переменную значением false и выполнив перебор элементов коллекции, проверяя каждый из них и присваивая var-переменной значение true, если будет найден предмет поиска. Метод, в котором такой подход используется для определения наличия в переданном списке List отрицательного числа, выглядит следующим образом:

```
def containsNeg(nums: List[Int]): Boolean = {
  var exists = false
  for (num <- nums)
    if (num < 0)
      exists = true
  exists
}
```

Если определить этот метод в интерпретаторе, то его можно вызвать следующими командами:

```
scala> containsNeg(List(1, 2, 3, 4))
res0: Boolean = false

scala> containsNeg(List(1, 2, -3, 4))
res1: Boolean = true
```

Но более лаконичный способ определения метода предусматривает вызов в отношении списка List функции высшего порядка exists:

```
def containsNeg(nums: List[Int]) = nums.exists(_ <
0)
```

Эта версия containsNeg выдает те же результаты, что и предыдущая:

```
scala> containsNeg(Nil)
```

```
res2: Boolean = false

scala> containsNeg(List(0, -1, -2))
res3: Boolean = true
```

Метод `exists` представляет собой управляющую абстракцию. Это специализированная циклическая конструкция, которая не встроена в язык Scala, как `while` или `for`, а предоставляется библиотекой Scala. В предыдущем разделе функция высшего порядка `filesMatching` позволила сократить повторяемость кода в реализации объекта `FileMatcher`. Метод `exists` обеспечивает такое же преимущество, но, поскольку это открытый метод в API коллекций Scala, сокращение повторяемости относится к клиентскому коду этого API. Если бы метода `exists` не было и потребовалось бы написать метод выявления наличия в списке четных чисел `containsOdd`, это можно было бы сделать следующим образом:

```
def containsOdd(nums: List[Int]): Boolean = {
  var exists = false
  for (num <- nums)
    if (num % 2 == 1)
      exists = true
  exists
}
```

Сравнивая тело метода `containsNeg` с телом метода `containsOdd`, можно заметить повторяемость во всем, за исключением условия проверки в выражении `expression`. С помощью метода `exists` вместо этого можно воспользоваться следующим кодом:

```
def containsOdd(nums: List[Int]) = nums.exists(_ %
2 == 1)
```

Тело кода в этой версии также практически идентично телу соответствующего метода `containsNeg` (той его версии, в которой используется `exists`), за исключением того, что условие, по которому выполняется поиск, иное. И тем не менее объем повторяющегося кода стал намного меньше, поскольку вся инфраструктура организации цикла убрана в метод `exists`.

В стандартной библиотеке Scala имеется множество других методов с организацией цикла. Как и `exists`, они зачастую могут сократить объем вашего кода, если появится возможность их применения.

9.3. Карринг

В главе 1 говорилось, что Scala позволяет создавать новые управляющие абстракции, которые воспринимаются как естественная языковая поддержка. Хотя показанные до сих пор примеры фактически и были управляющими абстракциями, вряд ли кто-то смог бы воспринять их как естественную поддержку со стороны языка. Чтобы понять, как создаются управляющие абстракции, больше похожие на расширения языка, сначала нужно разобраться с приемом функционального программирования, который называется *каррингом*.

Каррированная функция применяется не к одному, а к нескольким спискам аргументов. В листинге 9.2 показана обычная, некаррированная функция, складывающая два `Int`-параметра, `x` и `y`.

Листинг 9.2. Определение и вызов обычной функции

```
scala> def plainOldSum(x: Int, y: Int) = x + y
plainOldSum: (x: Int, y: Int)Int
scala> plainOldSum(1, 2)
res4: Int = 3
```

В листинге 9.3 показана аналогичная, но уже каррированная функция. Вместо списка из двух Int-параметров эта функция применяется к двум спискам, в каждом из которых содержится по одному Int-параметру.

Листинг 9.3. Определение и вызов каррированной функции

```
scala> def curriedSum(x: Int)(y: Int) = x + y
curriedSum: (x: Int)(y: Int)Int

scala> curriedSum(1)(2)
res5: Int = 3
```

Здесь при вызове curriedSum вы фактически получаете два обычных следующих непосредственно друг за другом вызова функции. Первый вызов функции получает единственный Int-параметр по имени x и возвращает функциональное значение для второй функции. Эта вторая функция получает Int-параметр y. Здесь действие функции по имени first соответствует тому, что должно было происходить при вызове первой традиционной функции curriedSum:

```
scala> def first(x: Int) = (y: Int) => x + y
first: (x: Int)Int => Int
```

Применение первой функции к числу 1, иными словами, вызов первой функции и передача ей значения 1, образует вторую функцию:

```
scala> val second = first(1)
second: Int => Int = <function1>
```

Применение второй функции к числу 2 дает результат

```
scala> second(2)
```

```
res6: Int = 3
```

Функции `first` и `second` являются всего лишь иллюстрациями процесса карринга. Они не связаны непосредственно с функцией `curriedSum`. И тем не менее это способ получить фактическую ссылку на вторую функцию из `curriedSum`. Чтобы воспользоваться `curriedSum` в выражении частично применяемой функции, можно использовать систему записи заместителя:

```
scala> val onePlus = curriedSum(1)_
onePlus: Int => Int = <function1>
```

Знак подчеркивания в `curriedSum(1)_` является заместителем для второго списка, используемого в качестве параметра[69]. В результате получается ссылка на функцию, при вызове которой единица прибавляется к ее единственному `Int`-аргументу, и возвращается результат

```
scala> onePlus(2)
res7: Int = 3
```

А вот как можно получить функцию, прибавляющую число 2 к ее единственному `Int`-аргументу:

```
scala> val twoPlus = curriedSum(2)_
twoPlus: Int => Int = <function1>

scala> twoPlus(2)
res8: Int = 4
```

9.4. Создание новых управляющих структур

В языках, использующих функции первого класса, даже если синтаксис языка устоялся, есть возможность эффективного

создания новых управляющих структур. Нужно лишь ввести методы, получающие функции в виде аргументов.

Например, в следующем фрагменте кода показана удваивающая управляющая структура — она повторяет операцию два раза и возвращает результат:

```
scala> def twice(op: Double => Double, x: Double)
= op(op(x))
twice: (op: Double => Double, x: Double)Double

scala> twice(_ + 1, 5)
res9: Double = 7.0
```

Типом `op` в данном примере является `Double => Double`. Это означает, что функция получает одно `Double`-значение в качестве аргумента и возвращает другое `Double`-значение.

При каждом обнаружении управляющей конструкции, повторяющейся во многих частях вашего кода, о ее реализации можно подумать как о новой управляющей структуре. Ранее в этой главе был показан узкоспециализированный шаблон управления `filesMatching`. Теперь давайте рассмотрим более широко применяющийся шаблон программирования: открытие ресурса, работа с ним, а затем закрытие ресурса. Все это можно собрать в управляющую абстракцию, воспользовавшись следующим методом:

```
def withPrintWriter(file: File, op: PrintWriter =>
Unit) = {
  val writer = new PrintWriter(file)
  try {
    op(writer)
  } finally {
    writer.close()
  }
```

```
}
```

При наличии такого метода им можно воспользоваться так:

```
withPrintWriter(
  new File("date.txt"),
  writer => writer.println(new java.util.Date)
)
```

Преимущества использования этого метода состоят в том, что закрытие файла в конце работы гарантируется `withPrintWriter`, а не пользовательским кодом. Поэтому забыть закрыть файл просто невозможно. Эта технология называется *шаблоном временного пользования* (loan pattern), поскольку функция управляющей абстракции, такая как `withPrintWriter`, открывает ресурс и отдает его функции во временное пользование. Например, в предыдущем примере `withPrintWriter` отдает во временное пользование `PrintWriter` функции `op`. Когда функция завершает свою работу, она сигнализирует, что ей уже не нужен одолженный ресурс. Затем в блоке `finally` ресурс закрывается, чем гарантируется его безусловное закрытие независимо от того, как завершилась работа функции, успешно или с выдачей исключения.

Один из способов придания клиентскому коду вида, делающего его похожим на встроенную управляющую структуру, предусматривает заключение списка аргументов в фигурные, а не круглые скобки. Если в Scala при каждом вызове метода ему передается строго один аргумент, можно заключить его не в круглые, а в фигурные скобки. Например, вместо:

```
scala> println("Hello, world!")
Hello, world!
```

можно написать:

```
scala> println { "Hello, world!" }
```

```
Hello, world!
```

Во втором примере аргумент для `println` вместо круглых скобок заключен в фигурные. Но такой прием использования фигурных скобок будет работать только при передаче одного аргумента.

Попытка нарушить это правило приводит к следующему результату:

```
scala> val g = "Hello, world!"
g: String = Hello, world!

Scala> g.substring { 7, 9 }
<console>:1: error: ';' expected but ',' found.
       g.substring { 7, 9 }
                     ^
```

Поскольку была предпринята попытка передачи функции `substring` двух аргументов, при их заключении в фигурные скобки выдается ошибка. Вместо фигурных в данном случае нужно использовать круглые скобки:

```
scala> g.substring(7, 9)
res12: String = wo
```

Возможность подстановки фигурных скобок вместо круглых для передачи одного аргумента обеспечивается программистам, использующим ваш код, с целью создания между фигурными скобками функционального литерала. Тем самым можно сделать вызов метода похожим на управляющую абстракцию. В качестве примера возьмем ранее определенный метод `withPrintWriter`. В своем самом последнем виде метод `withPrintWriter` получает два аргумента, поэтому использовать фигурные скобки нельзя. Тем не менее, поскольку функция, переданная `withPrintWriter`, является последним аргументом в списке, можно воспользоваться

каррингом, чтобы переместить первый аргумент типа `File` в отдельный список аргументов. Тогда функция останется единственным параметром второго списка параметров. Способ переопределения `withPrintWriter` показан в листинге 9.4.

Листинг 9.4. Использование шаблона временного пользования для записи в файл

```
def withPrintWriter(file: File)(op: PrintWriter =>
Unit) = {
  val writer = new PrintWriter(file)
  try {
    op(writer)
  } finally {
    writer.close()
  }
}
```

Отличие новой версии от старой состоит всего лишь в том, что теперь имеется два списка параметров, по одному параметру в каждом, а не один список из двух параметров. Посмотрите на пространство между двумя параметрами. В показанной здесь прежней версии `withPrintWriter` вы видите `...File, op...`. Но в этой версии вы видите `...File)(op...`. Благодаря определению, приведенному ранее, метод можно вызвать с применением более привлекательного синтаксиса:

```
val file = new File("date.txt")

withPrintWriter(file) { writer =>
  writer.println(new java.util.Date)
}
```

В этом примере первый список аргументов, в котором содержится один аргумент типа `File`, заключен в круглые скобки. А второй список аргументов, содержащий функциональный аргумент, заключен в фигурные скобки.

9.5. Параметры до востребования

Метод `withPrintWriter`, рассмотренный в предыдущем разделе, отличается от встроенных управляющих структур языка, таких как `if` и `while`, тем, что код между фигурными скобками получает аргумент. Функция, переданная `withPrintWriter`, требует одного аргумента типа `PrintWriter`. Этот аргумент показан в следующем коде как `writer =>`:

```
withPrintWriter(file) { writer =>
  writer.println(new java.util.Date)
}
```

А что нужно сделать, если понадобится реализовать что-нибудь больше похожее на `if` или `while`, где между фигурными скобками нет значения для передачи в код? Помочь справиться с подобными ситуациями могут имеющиеся в Scala *параметры до востребования* (by-name parameters).

Например, представим, что нужно реализовать конструкцию утверждения под названием `myAssert`[70]. Функция `myAssert` будет получать в качестве ввода функциональное значение и обращаться к флагу, чтобы решить, что делать. Если флаг установлен, `myAssert` вызовет переданную функцию и проверит, что она возвращает `true`. Если флаг сброшен, `myAssert` будет молча бездействовать.

Без использования параметров до востребования `myAssert` можно создать таким образом:

```
var assertionsEnabled = true
```

```
def myAssert(predicate: () => Boolean) =
  if (assertionsEnabled && !predicate())
    throw new AssertionError
```

С определением все в порядке, но пользоваться им неудобно:

```
myAssert(() => 5 > 3)
```

Конечно, лучше было бы обойтись в функциональном литерале без пустого списка параметров и обозначения `=>` и создать следующий код:

```
myAssert(5 > 3) // Не будет работать из-за
отсутствия () =>
```

Именно для того, чтобы можно было осуществить задуманное, и существуют параметры до востребования. Чтобы создать параметр до востребования, задавать его тип нужно с обозначения `=>`, а не с `() =>`. Например, можно заменить в `myAssert` параметр `predicate` параметром до востребования, изменив его тип `() => Boolean` на `=> Boolean`. Как это должно выглядеть, показано в листинге 9.5.

Листинг 9.5. Использование параметра до востребования

```
def byNameAssert(predicate: => Boolean) =
  if (assertionsEnabled && !predicate)
    throw new AssertionError
```

Теперь в свойстве, по поводу которого нужно высказать утверждение, от пустого параметра можно избавиться. В результате этого использование `byNameAssert` выглядит абсолютно так же, как встроенная управляющая структура:

```
byNameAssert(5 > 3)
```

Тип «до востребования» (by-name), в котором пустой список параметров `()` отбрасывается, допустимо использовать только в отношении параметров. Никаких by-name-переменных или by-name-полей не существует.

Можно, конечно, удивиться, почему нельзя просто написать функцию `myAssert`, воспользовавшись для ее параметров старым добрым типом `Boolean` и создав следующий код:

```
def boolAssert(predicate: Boolean) =
  if (assertionsEnabled && !predicate)
    throw new AssertionError
```

Разумеется, такая формулировка тоже сработает, и код, использующий эту версию `boolAssert`, будет выглядеть точно так же, как и прежде:

```
boolAssert(5 > 3)
```

И все же между этими двумя подходами есть одно заслуживающее внимания различие. Поскольку для параметра `boolAssert` используется тип `Boolean`, выражение внутри круглых скобок в `boolAssert(5 > 3)` вычисляется до вызова `boolAssert`. Выражение `5 > 3` выдает значение `true`, которое передается в `boolAssert`. В отличие от этого, поскольку типом параметра `predicate` функции `byNameAssert` является `=> Boolean`, выражение внутри круглых скобок в `byNameAssert(5 > 3)` до вызова `byNameAssert` не вычисляется. Вместо этого будет создано функциональное значение, чей метод `apply` вычислит `5 > 3`, и это функциональное значение будет передано функции `byNameAssert`.

Таким образом, разница между двумя подходами состоит в том, что при отключении утверждений вы увидите любые побочные эффекты, которые могут быть в выражении внутри круглых скобок

в boolAssert, но byNameAssert это не касается. Например, если утверждения отключены, попытки утверждать, что x / 0 == 0, в случае использования boolAssert приведут к выдаче исключения:

```
scala> val x = 5
x: Int = 5

scala> var assertionsEnabled = false
assertionsEnabled: Boolean = false

scala> boolAssert(x / 0 == 0)
java.lang.ArithmeticException: / by zero
  … 33 elided
```

Но попытки утверждать это на основе того же самого кода в случае использования byNameAssert не приведут к выдаче исключения:

```
scala> byNameAssert(x / 0 == 0)
```

Резюме

В этой главе было показано, как на основе имеющихся в Scala развитых функций создать поддержку для построения управляющих абстракций. Функции внутри вашего кода можно применять для избавления от распространенных шаблонов управления, а для многократного использования шаблонов управления, часто встречающихся в вашем программном коде, воспользуйтесь функциями высшего порядка из библиотеки Scala. Также были рассмотрены приемы использования карринга и параметров до востребования, позволяющих применять в весьма лаконичном синтаксисе собственные функции высшего порядка.

В двух последних главах было рассмотрено довольно много относящегося к функциям. В следующих нескольких главах мы

вернемся к дополнительным объектно-ориентированным средствам языка.

68 Такие специализированные методы циклической обработки определены в трейте Traversable, который является расширителем классов List, Set и Map. Более подробно этот вопрос рассматривается в главе 17.

69 В предыдущей главе, где система записи заместителя использовалась в отношении традиционных методов вроде println _, между именем и знаком подчеркивания нужно было ставить пробел. Здесь в этом нет необходимости, поскольку в Scala println_ является допустимым идентификатором, а curriedSum(1)_ — нет.

70 Здесь используется название myAssert, а не assert, поскольку имя assert предоставляется самим языком Scala. Соответствующее описание будет дано в разделе 14.1.

10. Композиция и наследование

В главе 6 была представлена часть основных объектно-ориентированных аспектов Scala. Данная глава подхватывает эту тему, продолжая с того места, где ее рассмотрение остановилось в главе 6, и дает углубленное и гораздо более полное представление об имеющейся в Scala поддержке объектно-ориентированного программирования.

Нам предстоит сравнить два основных вида взаимоотношений между классами: композицию и наследование. Композиция означает, что один класс содержит ссылку на другой и использует класс, на который ссылается, в качестве вспомогательного средства для выполнения своей миссии. Наследование представляет собой взаимоотношения родительского класса и его подкласса.

Кроме этих тем, будут рассмотрены абстрактные классы, методы без параметров, расширяемые классы, переопределяемые методы и поля, параметрические поля, вызов конструкторов родительского класса, полиморфизм и динамическое связывание, терминальные элементы и классы, а также фабричные объекты и методы.

10.1. Библиотека двумерной разметки

В качестве рабочего примера в этой главе создадим библиотеку для построения и вывода на экран двумерных элементов разметки. Каждый элемент будет представлен прямоугольником, заполненным текстом. Для удобства библиотека будет предоставлять фабричные методы по имени `elem`, создающие новые элементы из переданных данных. Например, вы сможете создать элемент разметки, содержащий строку, используя фабричный метод со следующей сигнатурой:

```
elem(s: String): Element
```

Как видите, элементы будут моделироваться с типом данных по имени `Element`. Для получения нового элемента, объединяющего два элемента, у вас появится возможность вызвать в отношении элемента операторы `above` или `beside`, передавая им второй элемент. Например, следующее выражение создаст более крупный элемент, содержащий два столбца, каждый высотой два элемента:

```
val column1 = elem("hello") above elem("***")
val column2 = elem("***") above elem("world")
column1 beside column2
```

Вывод результата этого выражения будет таким:

```
hello ***
*** world
```

Элементы разметки являются хорошим примером системы, в которой объекты могут создаваться из простых частей с помощью составляющих операторов. В этой главе мы определим классы, позволяющие составлять объекты элементов из массивов, рядов и прямоугольников. Эти объекты базовых элементов будут простыми деталями. Также определим составляющие операторы `above` и `beside`. Такие операторы зачастую называют комбинаторами, поскольку они объединяют элементы некой области в новые элементы.

Подходить к конструированию библиотеки лучше всего в понятиях комбинаторов: они позволяют осмыслить основные способы конструирования объектов в прикладной области. Что представляют собой простые объекты? Какими способами из простых объектов могут создаваться более интересные объекты? Как комбинаторы должны сочетаться друг с другом? Что должны представлять собой наиболее общие комбинации? Удовлетворяют ли они всем выдвигаемым правилам? Если на все эти вопросы вы отвечаете положительно, то вы на правильном пути.

10.2. Абстрактные классы

Нашей первой задачей будет определение типа `Element`, представляющего элементы разметки. Поскольку элементы являются двумерными прямоугольниками из символов, имеет смысл включить в класс метод по имени `contents`, ссылающийся на содержимое элемента разметки. Представлен `contents` может быть в виде массива строк, где каждая строка обозначает ряд. Отсюда типом возвращаемого `contents` значения должен быть `Array[String]`. Как он будет выглядеть, показано в листинге 10.1.

Листинг 10.1. Определение абстрактного метода и класса

```
abstract class Element {
  def contents: Array[String]
}
```

В этом классе `contents` объявляется в качестве метода, у которого нет реализации. Иными словами, метод является абстрактным элементом класса `Element`. Класс с абстрактными элементами сам по себе должен быть объявлен абстрактным, что делается указанием модификатора `abstract` перед ключевым словом `class`:

```
abstract class Element ...
```

Модификатор `abstract` указывает на то, что у класса могут быть абстрактные составляющие элементы, не имеющие реализации. Поэтому создать экземпляр абстрактного класса невозможно. При попытке сделать это будет получена ошибка компиляции:

```
scala> new Element
<console>:5: error: class Element is abstract;
  cannot be instantiated
```

```
    new Element
        ^
```

Чуть позже в этой главе мы рассмотрим способ создания подклассов класса `Element`, экземпляры которых можно будет создавать, поскольку они заполняют отсутствующее определение метода `contents`.

Обратите внимание на то, что у метода `contents` класса `Element` нет модификатора `abstract`. Метод является абстрактным, если у него нет реализации (например, знаков равенства или тела). В отличие от Java, здесь при объявлении методов не нужны или не разрешены модификаторы `abstract`. Методы, имеющие реализацию, называются конкретными.

И еще одно различие в терминологии между объявлениями и определениями. Класс `Element` *объявляет* абстрактный метод `contents`, но пока *не определяет* никаких конкретных методов. Но в следующем разделе класс `Element` будет усилен определением нескольких конкретных методов.

10.3. Определение методов без параметров

В качестве следующего шага к `Element` будут добавлены методы, выдающие ширину (`width`) и высоту (`height`) элемента (листинг 10.2). Метод `height` возвращает количество рядов в содержимом. Метод `width` возвращает длину первого ряда или, если рядов в элементе нет, нуль. Это означает, что нельзя определить элемент с нулевой высотой и ненулевой шириной.

Листинг 10.2. Определение не имеющих параметров методов width и height

```
abstract class Element {
  def contents: Array[String]
```

```
  def height: Int = contents.length
    def width: Int = if (height == 0) 0 else
contents(0).length
}
```

Заметьте, что ни в одном из трех методов класса `Element` нет списка параметров, даже пустого. Например, вместо:

```
def width(): Int
```

метод определен без круглых скобок:

```
def width: Int
```

Такие *методы без параметров* встречаются в Scala довольно часто. В отличие от них, методы, определенные с пустыми круглыми скобками, например `def height(): Int`, называются *методами с пустыми скобками*. Согласно имеющимся рекомендациям методы без параметров следует использовать, когда параметры отсутствуют и метод обращается к изменяемому состоянию только для чтения полей содержащего его объекта (при этом он не модифицирует изменяемое состояние). По этому соглашению поддерживается принцип унифицированного доступа[71], который гласит, что клиентский код не должен затрагиваться решением о реализации атрибута типа поля или метода.

Например, можно реализовать `width` и `height` в качестве полей, а не методов, просто заменив `def` в каждом определении на `val`:

```
abstract class Element {
  def contents: Array[String]
  val height = contents.length
  val width =
    if (height == 0) 0 else contents(0).length
```

```
}
```

С точки зрения клиента две пары определений абсолютно эквивалентны. Единственное различие заключается в том, что доступ к полю может осуществляться немного быстрее вызова метода, поскольку значения полей предварительно вычислены при инициализации класса, а не при каждом вызове метода. В то же время полям в каждом `Element`-объекте требуется дополнительное пространство памяти. Поэтому вопрос о том, как лучше представить атрибут, в виде поля или в виде метода, зависит от профиля пользователя класса, который со временем может измениться. Главное, чтобы на этот клиент класса `Element` никак не воздействовали изменения, вносимые во внутреннюю реализацию класса.

В частности, клиент класса `Element` не должен испытывать необходимости в перезаписи кода, если поле этого класса было переделано в функцию доступа, при условии, что это чисто функция доступа (то есть она не имеет никаких побочных эффектов и не зависит от изменяемого состояния). Как бы то ни было, клиент не должен решать какие-либо проблемы.

Пока у нас все получается. Но есть небольшое осложнение, связанное с методами работы Java. Дело в том, что в Java не реализуется принцип универсального доступа. Поэтому `string.length()` в Java — не то же самое, что `string.length`, и даже `array.length` — не то же самое, что `array.length()`. Вряд ли стоит говорить, насколько это усложняет дело.

Преодолеть это препятствие Scala помогает то, что он весьма мягко относится к смешиванию методов без параметров и методов с пустыми круглыми скобками. В частности, можно заменить метод без параметров методом с пустыми круглыми скобками и наоборот. Можно также не ставить пустые круглые скобки при вызове любой функции, не получающей аргументов. Например, в Scala одинаково допустимо применение двух следующих строк кода:

```
Array(1, 2, 3).toString
"abc".length
```

В принципе, в вызовах функций в Scala можно вообще не ставить пустые круглые скобки. Но их все же рекомендуется использовать, когда вызываемый метод представляет нечто большее, чем свойство своего объекта-получателя. Например, пустые круглые скобки уместны, если метод выполняет ввод-вывод, записывает переменные, которым может переприсваиваться значение (`var`-переменные), или считывает `var`-переменные, не являющиеся полями объекта-получателя, как непосредственно, так и косвенно путем использования изменяемых объектов. Таким образом, список параметров служит визуальным признаком того, что вызов инициирует некие примечательные вычисления, например:

```
"hello".length     // Нет (), поскольку побочные эффекты отсутствуют
println()          // Лучше () не отбрасывать
```

Подводя итоги, следует отметить, что в Scala приветствуется определение методов, не получающих параметров и не имеющих побочных эффектов, в виде методов без параметров, то есть без пустых круглых скобок. В то же время никогда не нужно определять метод, имеющий побочные эффекты, без круглых скобок, поскольку вызов этого метода будет выглядеть как выбор поля. Следовательно, ваши клиенты могут быть удивлены, столкнувшись с побочными эффектами.

По аналогии с этим при вызове функции, имеющей побочные эффекты, не забудьте при написании вызова поставить пустые круглые скобки. Есть еще один способ представления этого правила: если вызываемая функция выполняет операцию, используйте круглые скобки. Но если она просто предоставляет доступ к свойству, круглые скобки следует отбросить.

10.4. Расширение классов

Нам по-прежнему нужна возможность создавать объекты нового элемента. Вы уже видели, что новый класс `Element` не приспособлен для этого в силу своей абстрактности. Поэтому для получения экземпляра элемента необходимо создать подкласс, расширяющий класс `Element` и реализующий абстрактный метод `contents`. Как это делается, показано в листинге 10.3.

Листинг 10.3. Определение класса ArrayElement в качестве подкласса по отношению к классу Element

```
class ArrayElement(conts: Array[String]) extends
Element {
  def contents: Array[String] = conts
}
```

Класс `ArrayElement` определен с целью расширения класса `Element`. Как и в Java, для выражения данного обстоятельства после имени класса указывается уточнение `extends`:

```
… extends Element …
```

Использование уточнения `extends` заставляет класс `ArrayElement` унаследовать у класса `Element` все его незакрытые элементы и превращает тип `ArrayElement` в подтип относительно типа `Element`. Поскольку `ArrayElement` расширяет `Element`, класс `ArrayElement` называется подклассом класса `Element`. В свою очередь `Element` является родительским классом `ArrayElement`. Если не указать уточнение `extends`, компилятор Scala, безусловно, предположит, что ваш класс является расширением класса `scala.AnyRef`, который на платформе Java будет соответствовать классу `java.lang.Object`. Получается, класс `Element` неявно расширяет класс `AnyRef`. Эти отношения

наследования показаны на рис. 10.1.

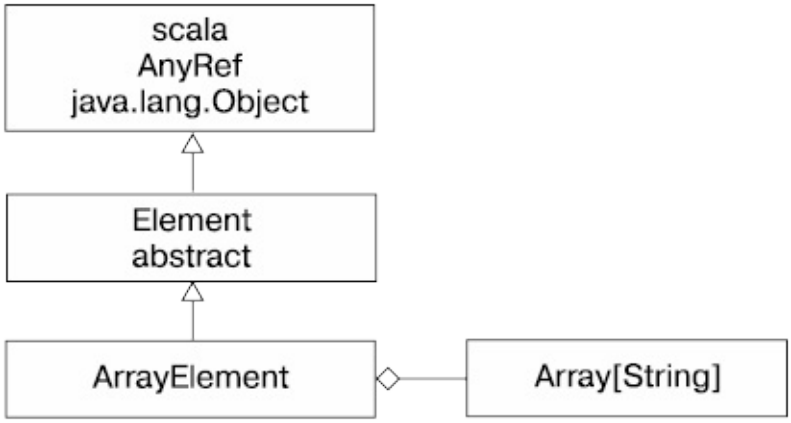


Рис. 10.1. Диаграмма классов для ArrayElement

Наследование означает, что все элементы родительского класса являются также элементами подкласса, но с двумя исключениями. Первое: закрытые элементы родительского класса не наследуются подклассом. Второе: элемент родительского класса не наследуется, если элемент с такими же именем и параметрами уже реализован в подклассе. В таком случае говорится, что элемент подкласса переопределяет элемент родительского класса. Если элемент в подклассе является конкретным, а в родительском классе абстрактным, также говорится, что конкретный элемент является реализацией абстрактного элемента.

Например, метод `contents` в `ArrayElement` переопределяет (или, в ином толковании, реализует) абстрактный метод `contents` в классе `Element`[72]. В отличие от этого, класс `ArrayElement` наследует у класса `Element` методы `width` и `height`. Например, располагая `ArrayElement`-объектом `ae`, можно запросить его ширину, используя выражение `ae.width`, как будто метод `width` был определен в классе `ArrayElement`:

```
scala> val ae = new ArrayElement(Array("hello",
"world"))
ae: ArrayElement = ArrayElement@39274bf7
```

```
scala> ae.width
res0: Int = 5
```

Подтипизация означает, что значение подкласса может быть использовано там, где требуется значение родительского класса. Например:

```
val e: Element = new ArrayElement(Array("hello"))
```

Переменная `e` определена как принадлежащая типу `Element`, следовательно, значение, используемое для ее инициализации, также должно быть типа `Element`. А фактически типом этого значения является `ArrayElement`. Это нормально, поскольку класс `ArrayElement` расширяет класс `Element`, и в результате тип `ArrayElement` совместим с типом `Element`[73].

На рис. 10.1 показаны взаимоотношения между `ArrayElement` и `Array[String]`, являющиеся *композицией*. Эти взаимоотношения называются композицией, поскольку класс `ArrayElement` состоит из класса `Array[String]`, то есть компилятор Scala помещает в создаваемый им для `ArrayElement` двоичный класс поле, содержащее ссылку на переданный массив `conts`.

Варианты, касающиеся композиции и наследования, будут рассмотрены чуть позже — в разделе 10.11.

10.5. Переопределение методов и полей

Принцип унифицированного доступа является одним из тех аспектов, где Scala подходит к полям и методам с единых позиций — не так, как Java. Еще одно отличие заключается в том, что в Scala поля и методы принадлежат одному и тому же пространству имен. Этот позволяет полю переопределить метод без параметров. Например, можно, как показано в листинге 10.4, изменить реализацию `contents` в классе `ArrayElement` из метода

в поле, не модифицируя определение абстрактного метода contents в классе Element:

Листинг 10.4. Переопределение метода без параметров в поле

```
class ArrayElement(conts: Array[String]) extends
Element {
  val contents: Array[String] = conts
}
```

Поле contents (определено с ключевым словом val) в этой версии ArrayElement является вполне подходящей реализацией метода без параметров contents (объявлено с ключевым словом def) в классе Element. В то же время в Scala запрещено в одном и том же классе определять поле и метод с одинаковыми именами, а в Java это разрешено.

Например, этот класс в Java пройдет компиляцию вполне успешно:

```
// Это код Java
class CompilesFine {
  private int f = 0;
  public int f() {
    return 1;
  }
}
```

Но соответствующий класс в Scala не откомпилируется:

```
class WontCompile {
   private var f = 0 // Не пройдет компиляцию,
поскольку поле
  def f = 1          // и метод имеют одинаковые
имена
```

```
}
```

В принципе, в Scala вместо четырех имеющихся в Java пространств имен для определений имеются только два пространства. Этими четырьмя пространствами имен в Java являются поля, методы, типы и пакеты. В отличие от этого, двумя пространствами имен в Scala являются:

- значения (поля, методы, пакеты и синглтон-объекты);
- типы (имена классов и трейтов).

Причина помещения в Scala полей и методов в одно и то же пространство имен заключается в предоставлении возможности переопределения методов без параметров в `val`-поля, чего нельзя сделать в Java[74].

10.6. Определение параметрических полей

Рассмотрим еще раз определение класса `ArrayElement`, показанное в предыдущем разделе. В нем имеется параметр `conts`, единственным предназначением которого является его копирование в поле `contents`. Имя `conts` было выбрано для параметра, чтобы оно походило на имя поля `contents`, но не вступало с ним в конфликт имен. Это код «с душком» — признак того, что в вашем коде может быть некая совершенно ненужная избыточность и повторяемость.

От этого кода сомнительного качества можно избавиться, скомбинировав параметр и поле в едином определении параметрического поля, что и показано в листинге 10.5.

Листинг 10.5. Определение contents в качестве параметрического поля

```
class ArrayElement(
  val contents: Array[String]
) extends Element
```

Обратите внимание на то, что параметр `contents` имеет префикс `val`. Это сокращенная форма записи, определяющая одновременно параметр и поле с одним и тем же именем. Если выразиться более конкретно, то класс `ArrayElement` теперь имеет поле `contents` (ему нельзя присвоить новое значение), доступ к которому может быть получен за пределами класса. Поле инициализировано значением параметра. Похоже на то, будто бы класс был написан следующим образом:

```
class ArrayElement(x123: Array[String]) extends
Element {
  val contents: Array[String] = x123
}
```

где `x123` является произвольным свежим именем для параметра.

Можно также поставить перед параметром класса префикс `var`, и тогда соответствующему полю можно будет переприсваивать значение. И наконец, подобным параметризованным полям, как и любым другим элементам класса, разрешено добавлять такие модификаторы, как `private`, `protected`[75] или `override`. Рассмотрим, к примеру, следующие определения классов:

```
class Cat {
  val dangerous = false
}
class Tiger(
  override val dangerous: Boolean,
  private var age: Int
) extends Cat
```

Определение класса Tiger является сокращенной формой для следующего альтернативного определения класса с переопределяемым элементом dangerous и закрытым элементом age:

```
class Tiger(param1: Boolean, param2: Int) extends
Cat {
  override val dangerous = param1
  private var age = param2
}
```

Оба элемента инициализируются из соответствующих параметров. Имена для этих параметров, param1 и param2, были выбраны произвольно. Главное, чтобы они не конфликтовали с какими-либо другими именами в пространстве имен.

10.7. Вызов конструкторов родительского класса

Теперь у вас имеется полноценная система из двух классов: абстрактного класса Element, который расширяется конкретным классом ArrayElement. Можно также наметить иные способы выражения элемента. Например, клиенту может понадобиться создать элемент разметки, содержащий один ряд, задаваемый строкой. Объектно-ориентированное программирование упрощает расширение системы новыми вариантами данных. Можно просто добавить подклассы. Например, в листинге 10.6 показан класс LineElement, расширяющий класс ArrayElement.

Листинг 10.6. Вызов конструктора родительского класса

```
class LineElement(s: String) extends
ArrayElement(Array(s)) {
  override def width = s.length
```

```
  override def height = 1
}
```

Поскольку `LineElement` расширяет `ArrayElement`, а конструктор `ArrayElement` получает параметр (`Array[String]`), `LineElement` нужно передать аргумент первичному конструктору своего родительского класса. Для вызова конструктора родительского класса аргумент или аргументы, которые нужно передать, просто помещаются в круглые скобки, стоящие за именем родительского класса. Например, класс `LineElement` передает аргумент `Array(s)` первичному конструктору класса `ArrayElement` путем помещения его в круглые скобки и указания после имени родительского класса `ArrayElement`:

```
... extends ArrayElement(Array(s)) ...
```

С появлением нового подкласса иерархия наследования для элементов разметки приобретает вид, показанный на рис. 10.2.

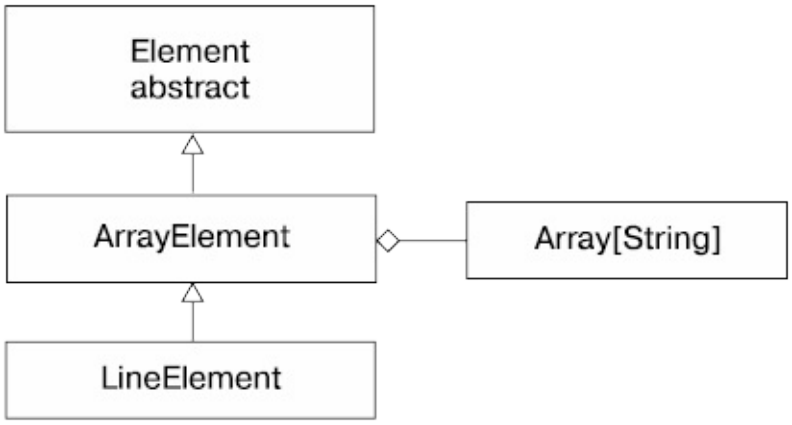


Рис. 10.2. Схема классов для LineElement

10.8. Использование модификаторов override

Обратите внимание на то, что определения `width` и `height` в `LineElement` имеют модификатор `override`. В разделе 6.3 этот модификатор встречался в определении метода `toString`. В Scala

такой модификатор требуется для всех элементов, переопределяющих конкретный элемент в родительском классе. Если элемент является реализацией абстрактного элемента с тем же именем, то модификатор указывать необязательно. Применять модификатор запрещено, если элемент не переопределяется или не является реализацией какого-либо другого элемента в родительском классе. Поскольку `height` и `width` в классе `LineElement` переопределяют конкретные определения в классе `Element`, модификатор `override` указывать обязательно.

Соблюдение этого правила дает полезную информацию для компилятора, которая помогает избежать некоторых трудноотлавливаемых ошибок и сделать развитие системы более безопасным. Например, если дать методу неправильное название или случайно указать для него не тот список параметров, компилятор тут же отреагирует выдачей сообщения об ошибке:

```
$ scalac LineElement.scala
.../LineElement.scala:50:
error: method hight overrides nothing
  override def hight = 1
               ^
```

Соглашение о применении модификатора `override` приобретает еще большее значение, когда дело доходит до развития системы. Скажем, вы определили библиотеку методов рисования двумерных фигур, сделали ее общедоступной, и она стала использоваться довольно широко. Посмотрев на эту библиотеку позже, вы захотели добавить к основному классу `Shape` новый метод с такой сигнатурой:

```
def hidden(): Boolean
```

Новый метод будет использоваться различными методами рисования для определения необходимости отрисовки той или иной фигуры. Тем самым можно существенно ускорить работу, но

сделать это, не рискуя вывести из строя клиентский код, невозможно. Кроме всего прочего, у клиента может быть определен подкласс класса `Shape` с другой реализацией `hidden`. Возможно, клиентский метод фактически заставит объект-получатель вместо проверки того, не является ли он невидимым, просто исчезнуть из виду. Поскольку две версии `hidden` переопределяют друг друга, ваши методы рисования в итоге заставят объекты исчезать, что совершенно не соответствует задуманному!

Эти случайные переопределения являются наиболее распространенным проявлением проблемы так называемого хрупкого основного класса. Суть ее заключается в том, что при добавлении в иерархии классов нового элемента к основным классам, которые мы обычно называем родительскими, создается риск вывести из строя клиентский код. Полностью разрешить проблему хрупкости основных классов в Scala невозможно, но по сравнению с Java ситуация улучшается[76]. Если библиотека рисования и ее клиентский код созданы в Scala, то у клиентской исходной реализации `hidden` не мог использоваться модификатор `override`, так как на момент его задействования не могло быть другого метода с таким же именем.

Поскольку во второй версии вашего класса фигур был добавлен метод `hidden`, при повторной компиляции кода клиента появится следующая ошибка:

```
.../Shapes.scala:6: error: error overriding method
    hidden in class Shape of type ()Boolean;
method hidden needs 'override' modifier
def hidden(): Boolean =
^
```

То есть вместо неверного поведения ваш клиент получит ошибку в ходе компиляции, что обычно является более предпочтительным исходом.

10.9. Полиморфизм и динамическое связывание

В разделе 10.4 было показано, что переменная типа `Element` может ссылаться на объект типа `ArrayElement`. Этот феномен называется *полиморфизмом*, что означает «множество образов» или «множество форм». В данном случае объекты типа `Element` могут иметь множество форм[77].

Ранее вам уже попадались две такие формы: `ArrayElement` и `LineElement`. Можно создать еще больше форм `Element`, определив новые подклассы класса `Element`. Например, можно определить новую форму `Element` с указанными шириной и высотой (`width` и `height`), заполняемую повсеместно заданным символом:

```
class UniformElement(
  ch: Char,
  override val width: Int,
  override val height: Int
) extends Element {
  private val line = ch.toString * width
  def contents = Array.fill(height)(line)
}
```

Иерархия наследования для класса `Element` теперь приобретает вид, показанный на рис. 10.3. В результате Scala примет все следующие присваивания, поскольку тип выражения присваивания соответствует типу определяемой переменной:

```
val e1: Element = new ArrayElement(Array("hello",
"world"))
val ae: ArrayElement = new LineElement("hello")
val e2: Element = ae
val e3: Element = new UniformElement('x', 2, 3)
```

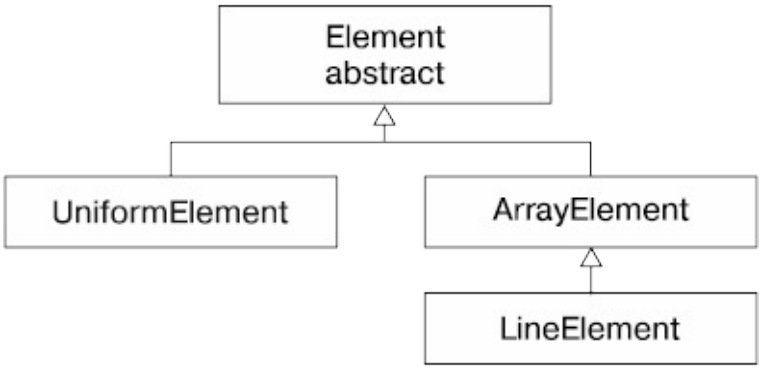


Рис. 10.3. Иерархия классов элементов разметки

Если изучить иерархию наследования, окажется, что в каждом их этих четырех `val`-определений тип выражения справа от знака равенства принадлежит типу `val`-переменной, инициализируемой слева от знака равенства.

Еще одна сторона вопроса касается динамического связывания вызовов методов с переменными и выражениями. Это означает, что текущая реализация вызываемого метода определяется на этапе выполнения программы на основе класса объекта, а не типа переменной или выражения. Чтобы продемонстрировать это поведение, мы временно уберем все существующие элементы из наших классов `Element` и добавим к `Element` метод по имени `demo`. Затем переопределим `demo` в `ArrayElement` и `LineElement`, но не в `UniformElement`:

```
abstract class Element {
  def demo() = {
    println("Element's implementation invoked")
  }
}

class ArrayElement extends Element {
  override def demo() = {
          println("ArrayElement's  implementation
invoked")
```

```
  }
}

class LineElement extends ArrayElement {
  override def demo() = {
            println("LineElement's   implementation
invoked")
  }
}

// UniformElement наследует demo из Element
class UniformElement extends Element
```

Если ввести данный код в интерпретатор, то можно будет определить метод, получающий объект типа `Element` и вызывающий в отношении этого объекта метод `demo`:

```
def invokeDemo(e: Element) = {
  e.demo()
}
```

Если передать методу `invokeDemo` объект типа `ArrayElement`, будет показано сообщение, свидетельствующее о вызове реализации `demo` из класса `ArrayElement`, даже если типом переменной `e`, в отношении которой был вызван метод `demo`, являлся `Element`:

```
scala> invokeDemo(new ArrayElement)
ArrayElement's implementation invoked
```

Аналогично, если передать `invokeDemo` объект типа `LineElement`, будет показано сообщение, свидетельствующее о вызове той реализации `demo`, которая была определена в классе `LineElement`:

```
scala> invokeDemo(new LineElement)
LineElement's implementation invoked
```

Поведение при передаче объекта типа `UniformElement` может на первый взгляд показаться неожиданным, но оно вполне корректно:

```
scala> invokeDemo(new UniformElement)
Element's implementation invoked
```

Поскольку в классе `UniformElement` метод `demo` не переопределяется, в нем наследуется реализация `demo` из его родительского класса `Element`. Таким образом, реализация, определенная в классе `Element`, будет той самой правильной реализацией для вызова `demo`, когда классом объекта является `UniformElement`.

10.10. Объявление терминальных элементов

Иногда при проектировании иерархии наследования нужно обеспечить невозможность переопределения элемента подклассом. В Scala, как и в Java, это делается путем добавления к элементу модификатора `final`. Как показано в листинге 10.7, модификатор `final` можно поставить для метода `demo` класса `ArrayElement`.

Листинг 10.7. Объявление терминального метода

```
class ArrayElement extends Element {
  final override def demo() = {
          println("ArrayElement's   implementation
invoked")
  }
}
```

При наличии данной версии в классе `ArrayElement` попытка переопределения `demo` в его подклассе `LineElement` не пройдет компиляцию:

```
elem.scala:18: error: error overriding method demo
  in class ArrayElement of type ()Unit;
method demo cannot override final member
  override def demo() = {
               ^
```

Порой может потребоваться обеспечить невозможность создания подкласса для класса в целом. Для этого нужно просто сделать весь класс терминальным, добавив к объявлению класса модификатор `final`. Например, в листинге 10.8 показано, как должен быть объявлен терминальным класс `ArrayElement`.

Листинг 10.8. Объявление терминального класса

```
final class ArrayElement extends Element {
  override def demo() = {
            println("ArrayElement's   implementation
invoked")
  }
}
```

При наличии данной версии класса `ArrayElement` любая попытка определения подкласса не пройдет компиляцию:

```
elem.scala:  18:  error:  illegal  inheritance  from
final class
    ArrayElement
  class LineElement extends ArrayElement {
                            ^
```

Теперь мы удалим модификаторы `final` и методы `demo` и вернемся к прежней реализации семейства классов `Element`. Далее в главе внимание будет сконцентрировано на завершении создания работоспособной версии библиотеки разметки.

10.11. Использование композиции и наследования

Композиция и наследование являются двумя способами определения нового класса в понятиях другого существующего класса. Если вы ориентируетесь преимущественно на многократное использование кода, то, как правило, предпочтение нужно отдавать композиции, а не наследованию. Проблемой хрупкости основного класса, вследствие которой можно ненароком сделать неработоспособными подклассы, внося изменения в родительский класс, страдает только наследование.

Насчет взаимоотношений наследования нужно задаться всего лишь одним вопросом: не моделируется ли ими взаимоотношение типа `is-a` (является частью)[78]? Например, нетрудно будет заметить, что класс `ArrayElement` является частью класса `Element`. Можно задаться еще одним вопросом: нужно ли будет клиентам использовать тип подкласса в качестве типа родительского класса[79]? Что касается класса `ArrayElement`, не вызывает никаких сомнений, что клиентам потребуется использовать объекты типа `ArrayElement` в качестве объектов типа `Element`.

Если задаться этими вопросами относительно взаимоотношений наследования, показанных на рис. 10.3, не покажутся ли вам какие-либо взаимоотношения подозрительными? В частности, насколько для вас очевидно, что `LineElement` является частью (`is-a`) `ArrayElement`? Как вы думаете, понадобится ли когда-нибудь клиентам тип `LineElement` в качестве типа `ArrayElement`?

Фактически класс LineElement был определен как подкласс класса ArrayElement преимущественно для многократного использования имеющегося в ArrayElement определения contents. Поэтому, возможно, будет лучше определить LineElement в качестве непосредственного подкласса класса Element:

```
class LineElement(s: String) extends Element {
  val contents = Array(s)
  override def width = s.length
  override def height = 1
}
```

В предыдущей версии у LineElement имелись взаимоотношения наследования с ArrayElement, откуда он наследовал метод contents. Теперь же у него отношения композиции с классом Array: в нем содержится ссылка на строковый массив из его собственного поля contents[80]. При наличии этой реализации класса LineElement иерархия наследования для Element приобретет вид, показанный на рис. 10.4.

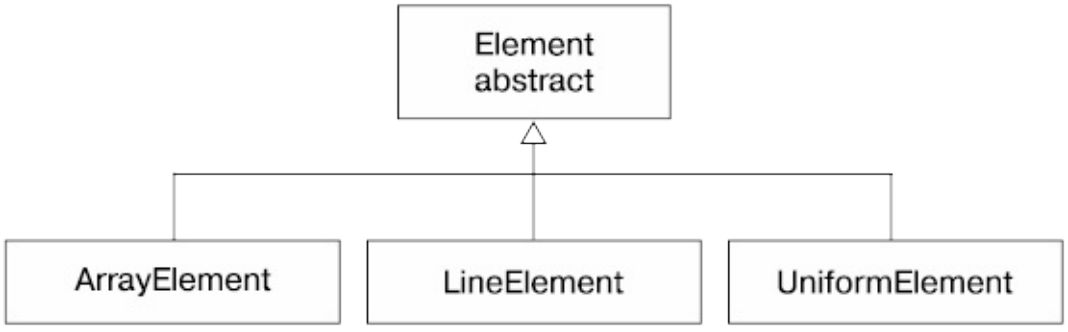


Рис. 10.4. Иерархия класса с пересмотренным определением подкласса LineElement

10.12. Реализация above, beside и toString

В качестве следующего шага в классе Element будет реализован метод above. Помещение одного элемента выше другого с помощью метода above означает объединение двух значений содержимого элементов, представленного contents, в единое целое. Поэтому первый, черновой вариант метода above может иметь следующий вид:

```
def above(that: Element): Element =
  new ArrayElement(this.contents ++ that.contents)
```

Операция ++ объединяет два массива. Массивы в Scala представлены в виде массивов Java, но поддерживают значительно большее количество методов. В частности, массивы в Scala могут быть преобразованы в экземпляры класса scala.Seq, которые представляют собой структуры, похожие на последовательности, и содержат несколько методов для доступа к последовательностям и преобразования этих последовательностей. В этой главе будут рассмотрены и некоторые другие методы, а развернутое представление о них дано в главе 17.

В действительности показанный ранее код нельзя считать подходящим по всем статьям, поскольку он не позволяет помещать друг на друга элементы разной ширины. Но, чтобы в этом разделе ничего не усложнять, оставим все как есть и станем передавать above только элементы одинаковой длины. В разделе 10.14 мы усовершенствуем above, чтобы клиенты могли использовать этот метод для объединения элементов разной ширины.

Следующим будет реализован метод beside. Чтобы поставить элементы рядом друг с другом, создадим новый элемент, в котором каждый ряд будет получен путем объединения соответствующих рядов двух элементов. Как и прежде, чтобы ничего не усложнять, начнем с предположения, что высота двух элементов одинакова. Тогда конструкция метода beside приобретет следующий вид:

```
def beside(that: Element): Element = {
        val    contents    =    new    Array[String]
(this.contents.length)
  for (i <- 0 until this.contents.length)
            contents(i)   =   this.contents(i)   +
that.contents(i)
  new ArrayElement(contents)
}
```

Метод `beside` сначала назначает новый массив `contents` и заполняет его объединением соответствующих элементов из массивов `this.contents` и `that.contents`. В итоге получается новый `ArrayElement`, имеющий новое содержимое `contents`.

Хотя эта реализация `beside` вполне работоспособна, в ней используется императивный стиль программирования, о чем явно свидетельствует наличие цикла, обходящего элементы массивов по индексам. В альтернативном варианте метод можно сократить до одного выражения:

```
new ArrayElement(
  for (
          (line1,   line2)   <-   this.contents   zip
that.contents
  ) yield line1 + line2
)
```

Здесь с помощью оператора `zip` массивы `this.contents` и `that.contents` преобразуются в массив пар (при вызовах `Tuple2`). Оператор `zip` выбирает соответствующие элементы в двух своих операндах и формирует массив пар. Например, выражение

```
Array(1, 2, 3) zip Array("a", "b")
```

будет вычислено в:

```
Array((1, "a"), (2, "b"))
```

Если один из двух массивов-операндов длиннее другого, оставшиеся элементы оператор `zip` просто отбрасывает. В показанном ранее выражении третий элемент левого операнда, `3`, не формирует пару результата, поскольку для него не находится соответствующего элемента в правом операнде.

Затем выполняется обход элементов объединенного массива с помощью выражения. Здесь синтаксис `for ((line1, line2) <- ...)` позволяет указать имена обоих элементов пары в одном *шаблоне* (то есть теперь `line1` обозначает первый элемент пары, а `line2` — второй). Имеющаяся в Scala система поиска по шаблону будет подробнее рассмотрена в главе 15. А сейчас все это можно представить себе как способ определения для каждого шага итерации двух `val`-переменных, `line1` и `line2`.

У выражения `for` есть составная часть `yield`, и поэтому оно выдает результат. Этот результат того же вида, что и перебираемый выражением объект (то есть данный массив). Каждый элемент является результатом объединения соответствующих рядов, `line1` и `line2`. Следовательно, конечный результат выполнения этого кода получается таким же, как и результат выполнения первой версии `beside`, но, поскольку в нем удалось обойтись без явной индексации массива, результат достигается способом, при котором допускается меньше ошибок.

Но вам все еще нужен способ отображения элементов. Как обычно, отображение выполняется с помощью определения метода `toString`, возвращающего элемент, отформатированный в виде строки. Его определение выглядит следующим образом:

```
override def toString = contents mkString "\n"
```

В реализации `toString` используется метод `mkString`, который определен для всех последовательностей, включая массивы. Как было показано в разделе 7.8, такое выражение, как

`arr mkString sep`, возвращает строку, состоящую из всех элементов массива `arr`. Каждый элемент отображается на строку путем вызова его метода `toString`. Между последовательными элементами строк вставляется строка-разделитель `sep`. Таким образом, выражение `contents mkString "\n"` форматирует содержимое массива как строку, где каждый элемент массива появляется в своем собственном ряду.

Обратите внимание на то, что метод `toString` не требует указания пустого списка параметров. Это соответствует рекомендациям по соблюдению принципа унифицированного доступа, поскольку `toString` является чистым методом, не получающим никаких параметров. После добавления этих трех методов класс `Element` приобретет вид, показанный в листинге 10.9.

Листинг 10.9. Класс Element с методами above, beside и toString

```
abstract class Element {
  def contents: Array[String]

  def width: Int =
    if (height == 0) 0 else contents(0).length

  def height: Int = contents.length

  def above(that: Element): Element =
            new   ArrayElement(this.contents   ++
that.contents)

  def beside(that: Element): Element =
    new ArrayElement(
      for (
```

```
            (line1, line2) <- this.contents zip
that.contents
      ) yield line1 + line2
    )

  override def toString = contents mkString "\n"
}
```

10.13. Определение фабричного объекта

Теперь у вас есть иерархия классов для элементов разметки. Можно предоставить ее вашим клиентам как есть или выбрать технологию скрытия иерархии за фабричным объектом.

В фабричном объекте содержатся методы, конструирующие другие объекты. Затем эти фабричные методы будут использоваться клиентами для конструирования объектов вместо того, чтобы конструировать объекты непосредственно с помощью ключевого слова `new`. Преимущество такого подхода заключается в возможности централизации создания объектов и в скрытии способа представления объектов с помощью классов. Такое скрытие не только сделает вашу библиотеку понятнее для клиентов, поскольку в открытом виде будет представлено меньше подробностей, но и обеспечит вам больше возможностей для последующего изменения реализации библиотеки без нарушения работы клиентского кода.

Первой задачей при конструировании фабрики для элементов разметки является выбор места, в котором должны располагаться фабричные методы. Чьими элементами они должны быть, синглтон-объекта или класса? Как должен быть назван содержащий их объект или класс? Существует множество возможностей. Самое простое решение заключается в создании объекта-спутника класса `Element` и превращении его в фабричный объект для элементов разметки. Таким образом, клиентам нужно предоставить только

комбинацию «класс — объект Element», а реализацию дерева классов ArrayElement, LineElement и UniformElement можно скрыть.

В листинге 10.10 представлена конструкция объекта Element, которая соответствует этой схеме. В объекте Element содержатся три переопределяемых варианта метода elem, которые конструируют различный вид объекта разметки.

Листинг 10.10. Фабричный объект с фабричными методами

```
object Element {

  def elem(contents: Array[String]): Element =
    new ArrayElement(contents)

   def elem(chr: Char, width: Int, height: Int):
Element =
    new UniformElement(chr, width, height)

  def elem(line: String): Element =
    new LineElement(line)
}
```

С появлением этих фабричных методов наметился смысл изменить реализацию класса Element таким образом, чтобы в нем вместо явного создания новых экземпляров ArrayElement-объектов выполнялись фабричные методы elem. Для вызова фабричных методов без указания с ними имени синглтон-объекта Element мы импортируем в верхней части кода исходный файл Element.elem. Иными словами, вместо вызова фабричных методов с помощью указания Element.elem внутри класса Element мы импортируем Element.elem, чтобы можно было просто вызвать фабричные методы по имени elem. Код класса

Element после внесения изменений показан в листинге 10.11.

Листинг 10.11. Класс Element, реорганизованный для использования фабричных методов

```
import Element.elem

abstract class Element {

  def contents: Array[String]

  def width: Int =
    if (height == 0) 0 else contents(0).length

  def height: Int = contents.length

  def above(that: Element): Element =
    elem(this.contents ++ that.contents)

  def beside(that: Element): Element =
    elem(
      for (
            (line1, line2) <- this.contents zip
that.contents
      ) yield line1 + line2
    )

  override def toString = contents mkString "\n"
}
```

Кроме того, благодаря наличию фабричных методов подклассы ArrayElement, LineElement и UniformElement теперь могут

стать закрытыми, поскольку отпадет надобность непосредственного обращения к ним со стороны клиентов. В Scala классы и синглтон-объекты можно определять внутри других классов и синглтон-объектов. Один из способов превращения подклассов класса `Element` в закрытые заключается в помещении их внутрь синглтон-объекта `Element` и объявлении их там закрытыми. Классы по-прежнему будут доступны трем фабричным методам `elem` там, где в них есть надобность. Как это будет выглядеть, показано в листинге 10.12.

Листинг 10.12. Скрытие реализации с помощью использования закрытых классов

```
object Element {

  private class ArrayElement(
    val contents: Array[String]
  ) extends Element

   private class LineElement(s: String) extends
Element {
    val contents = Array(s)
    override def width = s.length
    override def height = 1
  }

  private class UniformElement(
    ch: Char,
    override val width: Int,
    override val height: Int
  ) extends Element {
    private val line = ch.toString * width
```

```
    def contents = Array.fill(height)(line)
  }

  def elem(contents: Array[String]): Element =
    new ArrayElement(contents)

   def elem(chr: Char, width: Int, height: Int):
Element =
    new UniformElement(chr, width, height)

  def elem(line: String): Element =
    new LineElement(line)
}
```

10.14. Методы heighten и widen

Нам нужно внести еще одно, последнее усовершенствование. Версия `Element`, показанная в листинге 10.11, устроить нас всецело не может, поскольку она не позволяет клиентам помещать друг на друга элементы разной ширины или рядом друг с другом элементы разной высоты.

Например, вычисление следующего выражения не будет работать корректно, потому что второй ряд в объединенном элементе длиннее первого:

```
new ArrayElement(Array("hello")) above
new ArrayElement(Array("world!"))
```

Аналогично этому вычисление следующего выражения не будет работать правильно из-за того, что высота первого элемента `ArrayElement` составляет два ряда, а второго — только один:

```
new ArrayElement(Array("one", "two")) beside
new ArrayElement(Array("one"))
```

В листинге 10.13 показан закрытый вспомогательный метод по имени `widen`, который получает ширину и возвращает `Element`-объект такой ширины. Результат включает в себя содержимое этого `Element`-объекта, которое для достижения нужной ширины отцентрировано за счет создания отступов справа и слева с применением любого нужного для этого количества пробелов. В листинге 10.13 также показан похожий метод `heighten`, выполняющий ту же функцию в вертикальном направлении. Метод `widen` вызывается методом `above`, чтобы обеспечить одинаковую ширину элементов, помещаемых друг над другом. Аналогично этому метод `heighten` вызывается методом `beside`, чтобы обеспечить одинаковую высоту элементов, помещаемых рядом друг с другом. После внесения этих изменений библиотека разметки будет готова к использованию.

Листинг 10.13. Класс Element с методами widen и heighten

```
import Element.elem

abstract class Element {
  def contents: Array[String]

  def width: Int = contents(0).length
  def height: Int = contents.length

  def above(that: Element): Element = {
    val this1 = this widen that.width
    val that1 = that widen this.width
    elem(this1.contents ++ that1.contents)
  }

  def beside(that: Element): Element = {
```

```
    val this1 = this heighten that.height
    val that1 = that heighten this.height
    elem(
        for ((line1, line2) <- this1.contents zip
that1.contents)
      yield line1 + line2)
  }

  def widen(w: Int): Element =
    if (w <= width) this
    else {
         val left = elem(' ', (w - width) / 2,
height)
           val right = elem(' ', w - width -
left.width, height)
      left beside this beside right
    }

  def heighten(h: Int): Element =
    if (h <= height) this
    else {
      val top = elem(' ', width, (h - height) / 2)
        val bot = elem(' ', width, h - height -
top.height)
      top above this above bot
    }

  override def toString = contents mkString "\n"
}
```

10.15. А теперь соберем все вместе

Интересным способом применения почти всех элементов библиотеки разметки будет написание программы, рисующей спираль с заданным количеством ребер. Ее созданием займется программа `Spiral`, показанная в листинге 10.14.

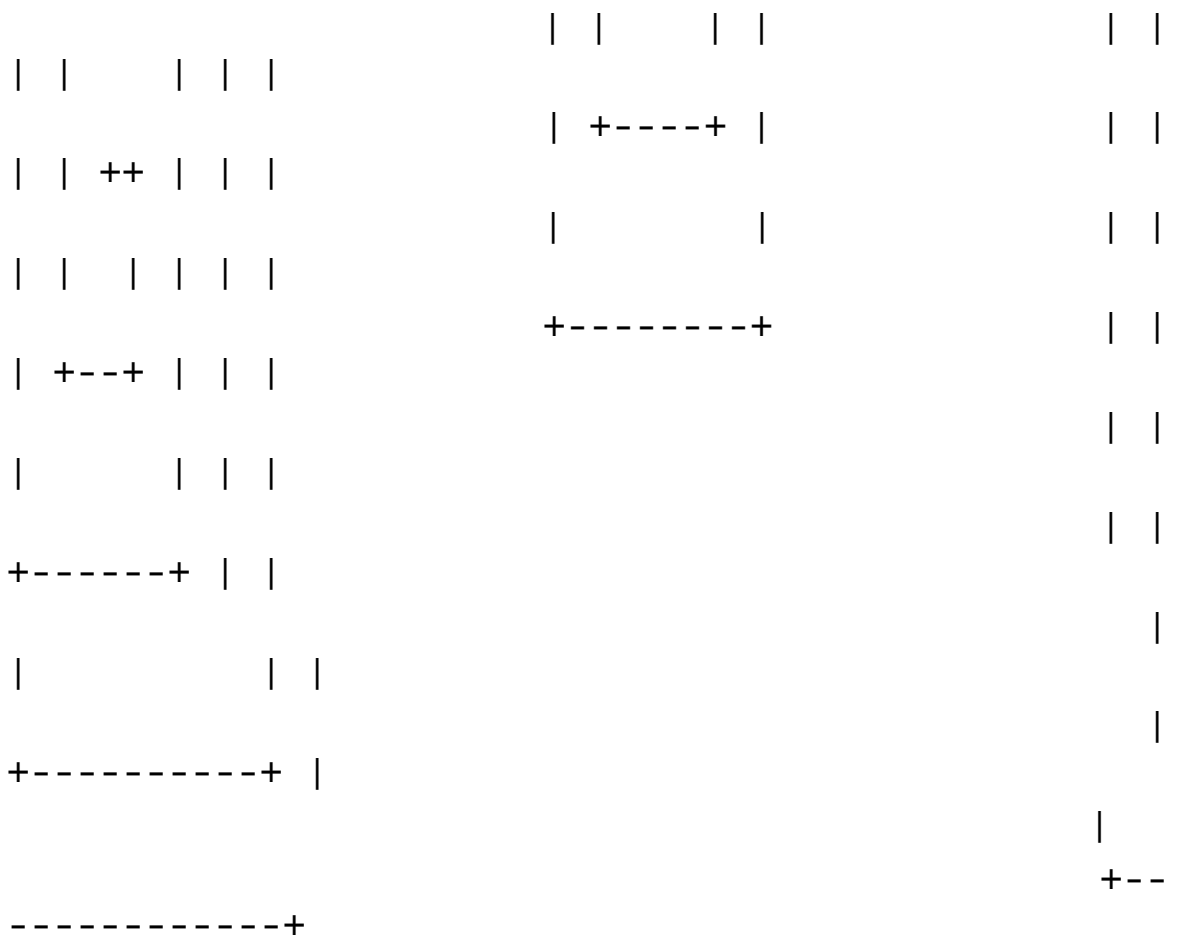
Листинг 10.14. Приложение Spiral

```
import Element.elem

object Spiral {

  val space = elem(" ")
  val corner = elem("+")

  def spiral(nEdges: Int, direction: Int): Element
= {
    if (nEdges == 1)
      elem("+")
    else {
       val sp = spiral(nEdges - 1, (direction + 3)
% 4)
      def verticalBar = elem('|', 1, sp.height)
      def horizontalBar = elem('-', sp.width, 1)
      if (direction == 0)
          (corner beside horizontalBar) above (sp
beside space)
      else if (direction == 1)
           (sp above space) beside (corner above
verticalBar)
      else if (direction == 2)
           (space beside sp) above (horizontalBar
beside corner)
      else
```

```
          (verticalBar above corner) beside (space
above sp)
    }
  }

  def main(args: Array[String]) = {
    val nSides = args(0).toInt
    println(spiral(nSides, 0))
  }
}
```

Поскольку `Spiral` является автономным объектом с методом `main`, имеющим надлежащую сигнатуру, этот код можно считать приложением, написанным на Scala. `Spiral` получает один аргумент командной строки в виде целого числа и рисует спираль с указанным числом граней. Например, можно нарисовать шестигранную спираль, как показано слева, и более крупную спираль, как показано справа.

```
$ scala Spiral 6        $ scala Spiral 11        $
scala Spiral 17
+-----                  +----------              +--
--------------
|                      |                        |
| +-+                    | +------+                 |
+------------+
| + |                    | |      |                 |
|            |
|   |                    | | +--+ |                 | |
+--------+ |
+---+                  | | |  | |                 | |
|        | |
                         | | ++ | |                 | |
| +----+ | |
```



Резюме

В этой главе были рассмотрены дополнительные концепции, относящиеся к объектно-ориентированному программированию на языке Scala. Среди них представлены абстрактные классы, наследование, создание подтипов, иерархии классов, параметрические поля и переопределение методов. У вас должно было выработаться понимание способов создания в Scala оригинальных иерархий классов. А к работе с библиотекой раскладки мы еще вернемся в главе 14.

71 *Meyer B.* Object-Oriented Software Construction. — Prentice Hall, 2000.

72 Один из недостатков этой конструкции заключается в том, что из-за изменяемости возвращаемого массива клиенты могут его модифицировать. В книге мы старались ничего не усложнять, но будь ArrayElement частью реального проекта, можно было бы рассмотреть

возвращение вместо этого копии массива, позволяющей защититься от изменений. Еще одна проблема заключается в том, что мы пока не гарантируем одинаковой длины массива contents каждого String-элемента. Задача может быть решена путем проверки соблюдения предварительного условия в первичном конструкторе и выдачи исключения при его нарушении.

73 Чтобы получить более четкое представление о разнице между подклассом и подтипом, обратитесь к словарной статье о подтипах.

74 Причиной того, что пакеты в Scala используют общее с полями и методами пространство имен, является стремление предоставить вам возможность получать доступ к импорту пакетов (а не только к именам типов), а также к полям и методам синглтон-объектов. Это также входит в перечень того, что невозможно сделать в Java. Подробности будут рассмотрены в разделе 13.3.

75 Модификатор protected, предоставляющий доступ к подклассам, будет подробно рассмотрен в главе 13.

76 В Java 1.5 была введена аннотация @Override, работающая аналогично имеющемуся в Scala модификатору override, но, в отличие от Scala-модификатора override, ее применение не является обязательным.

77 Эта разновидность полиморфизма называется полиморфизмом подтипирования. Еще одна разновидность полиморфизма в Scala, которая называется универсальным полиморфизмом, рассматривается в главе 19.

78 *Meyers S.* Effective C++. — Addison-Wesley, 1991.

79 *Eckel B.* Thinking in Java. — Prentice Hall, 1998.

80 Класс ArrayElement также имеет взаимоотношения композиции с классом Array, поскольку его параметрическое поле contents содержит ссылку на строковый массив. Код для ArrayElement показан в листинге 10.5. Его взаимоотношения композиции представлены в диаграмме классов в виде ромба, к примеру на рис. 10.1.

11. Иерархия в Scala

После изучения в предыдущей главе подробностей наследования классов самое время немного отступить назад и посмотреть на иерархию классов в Scala в целом. В Scala каждый класс наследуется из общего родительского класса по имени Any. Поскольку каждый класс является подклассом Any, методы, определенные в классе Any, являются универсальными: их можно вызвать в отношении любого объекта. В самом низу иерархии в Scala также определяются довольно интересные классы Null и Nothing, которые, по сути, выступают в роли общих подклассов. Например, в то время как Any является родительским классом для всех прочих классов, Nothing — подкласс для любого иного класса. В данной главе вам будет предоставлен тур по имеющейся в Scala иерархии классов.

11.1. Иерархия классов в Scala

На рис. 11.1 показана схема иерархии классов в Scala. На вершине иерархии находится класс Any, в котором определяются методы, в число которых входят:

```
final def ==(that: Any): Boolean
final def !=(that: Any): Boolean
def equals(that: Any): Boolean
def ##: Int
def hashCode: Int
def toString: String
```

Поскольку все классы являются наследниками класса Any, в отношении каждого объекта в программе на Scala может быть проведено сравнение с использованием методов ==, != или equals, хеширование с использованием методов ## или hashCode

и форматирование с использованием метода toString. Методы определения равенства и неравенства == и != объявлены в классе Any терминальными, следовательно, переопределить их в подклассах невозможно. Метод == — по сути, то же самое, что и метод equals, а метод != всегда является отрицанием метода equals[81]. Следовательно, отдельные классы могут перекроить смысл значения метода == или != путем переопределения метода equals. Соответствующий пример будет показан в этой главе чуть позже.

У корневого класса Any имеются два подкласса: AnyVal и AnyRef. Класс AnyVal является родительским классом для классов значений в Scala. Наряду с возможностью определения собственных классов значений (см. раздел 11.4) Scala имеет девять встроенных классов значений: Byte, Short, Char, Int, Long, Float, Double, Boolean и Unit. Первые восемь из них соответствуют элементарным типам, имеющимся в Java, и их значения на этапе выполнения программы представляются в виде элементарных значений Java. Все экземпляры этих классов написаны в Scala в виде литералов. Например, 42 является экземпляром класса Int, 'x' — экземпляром Char, а false — экземпляром класса Boolean. Создать экземпляры этих классов, используя ключевое слово new, невозможно. Это обеспечивается за счет особого приема, заключающегося в том, что все классы значений определены и абстрактными, и терминальными одновременно.

Поэтому, если воспользоваться следующим кодом:

```
scala> new Int
```

будет получен такой результат:

```
<console>:5: error: class Int is abstract; cannot
be
instantiated
```

```
        new Int
        ^
```

Еще один класс значений, `Unit`, примерно соответствует имеющемуся в Java типу `void` — он используется в качестве типа результата выполнения метода, который не возвращает содержательного результата. Как упоминалось в разделе 7.2, у `Unit` имеется единственное значение экземпляра, которое записывается как `()`.

В соответствии с объяснениями, изложенными в главе 5, в классах значений в качестве методов поддерживаются обычные арифметические и логические (булевы) операторы. Например, у класса `Int` имеются методы `+` и `*`, а у класса `Boolean` — методы `||` и `&&`. Классы значений также наследуют все методы из класса `Any`. Это можно протестировать в интерпретаторе:

```
scala> 42.toString
res1: String = 42

scala> 42.hashCode
res2: Int = 42

scala> 42 equals 42
res3: Boolean = true
```

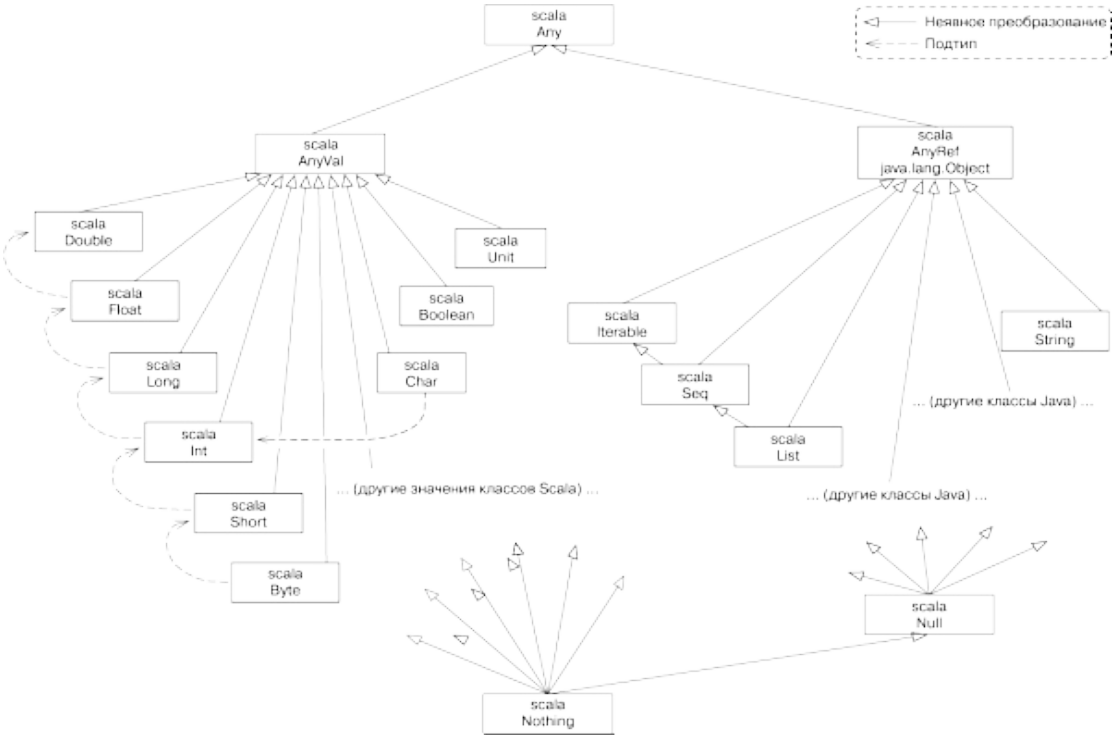


Рис. 11.1. Иерархия классов в Scala

Следует заметить, что пространство классов значений плоское — все классы значений являются подтипами `scala.AnyVal`, но не являются подклассами друг друга. Вместо этого между различными типами классов значений существует подразумеваемое преобразование типов. Например, экземпляр класса `scala.Int`, когда это требуется, автоматически расширяется (путем подразумеваемого преобразования) в экземпляр класса `scala.Long`.

Как упоминалось в разделе 5.10, подразумеваемое преобразование используется также для добавления дополнительных функциональных возможностей к типам значений. Например, тип `Int` поддерживает все показанные далее операции:

```
scala> 42 max 43
res4: Int = 43
```

```
scala> 42 min 43
res5: Int = 42

scala> 1 until 5
res6: scala.collection.immutable.Range = Range(1,
2, 3, 4)

scala> 1 to 5
res7: scala.collection.immutable.Range.Inclusive
  = Range(1, 2, 3, 4, 5)

scala> 3.abs
res8: Int = 3

scala> (-3).abs
res9: Int = 3
```

Работает это следующим образом: все методы `min`, `max`, `until`, `to` и `abs` определены в классе `scala.runtime.RichInt`, а между классами `Int` и `RichInt` существует подразумеваемое преобразование. Оно применяется при вызове в отношении `Int`-объекта метода, который определен не в классе `Int`, а в классе `RichInt`. По аналогии с этим классы-усилители и подразумеваемое преобразование существуют и для других классов значений. Более подробно подразумеваемые преобразования будут рассматриваться в главе 21.

Другим подклассом корневого класса `Any` является класс `AnyRef`. Это базовый класс всех *ссылочных классов* в Scala. Как упоминалось ранее, на платформе Java `AnyRef` фактически является псевдонимом класса `java.lang.Object`. Следовательно, все классы, написанные на Java и Scala, являются наследниками класса `AnyRef`[82]. Поэтому одним из способов составления собственного представления о `java.lang.Object` является способ

реализации AnyRef на платформе Java. Следовательно, хотя Object и AnyRef можно взаимозаменяемо использовать в программах Scala на платформе Java, рекомендуемым стилем будет повсеместное использование AnyRef.

11.2. Реализация элементарных типов

А как все это реализовано? Фактически в Scala целочисленные значения хранятся так же, как и в Java, — в виде 32-разрядных слов. Это необходимо для эффективной работы виртуальной машины Java (JVM), а также для обеспечения возможности совместной работы с библиотеками Java. Такие стандартные операции, как сложение или умножение, реализуются в качестве элементарных операций. Но как только целочисленное значение в Scala нужно представлять в виде объекта Java, используется дублирующий класс java.lang.Integer. Так, к примеру, происходит при вызове в отношении целого числа метода toString или присваивании целого числа переменной типа Any. При необходимости целочисленные значения типа Int явно преобразуются в упакованные целые числа типа java.lang.Integer.

Это во многом походит на автоупаковку (auto-boxing) в Java 5, оба процесса действительно очень похожи. Но все-таки есть одно коренное различие: упаковка в Scala менее заметна, чем упаковка в Java. Попробуйте выполнить в Java следующий код:

```
// Это код на языке Java
boolean isEqual(int x, int y) {
  return x == y;
}
System.out.println(isEqual(421, 421));
```

В результате, конечно же, будет получено значение true. А теперь измените типы аргументов isEqual на

java.lang.Integer (или с аналогичным результатом на Object):

```
// Это код на языке Java
boolean isEqual(Integer x, Integer y) {
  return x == y;
}
System.out.println(isEqual(421, 421));
```

Получите результат false! Оказывается, упаковка числа 421 была выполнена дважды, поэтому аргументами для x и y стали два разных объекта. Поскольку применение == в отношении ссылочных типов означает эквивалентность ссылок, а Integer является ссылочным типом, в результате получается false. Это один из аспектов, свидетельствующий о том, что Java не является чистым объектно-ориентированным языком. Существует четко прослеживаемая разница между элементарными и ссылочными типами.

Теперь попробуйте провести тот же самый эксперимент на Scala:

```
scala> def isEqual(x: Int, y: Int) = x == y
isEqual: (x: Int, y: Int)Boolean


scala> isEqual(421, 421)
res10: Boolean = true

scala> def isEqual(x: Any, y: Any) = x == y
isEqual: (x: Any, y: Any)Boolean


scala> isEqual(421, 421)
res11: Boolean = true
```

Операция равенства в Scala разработана так, чтобы быть понятной относительно представления типов. Для типов значений (числовых или логических) это вполне естественное равенство. Для ссылочных типов, отличающихся от упакованных числовых типов Java, `==` рассматривается в качестве псевдонима метода `equals`, унаследованного от класса `Object`. Этот метод изначально определен для выявления эквивалентности ссылок, но во многих подклассах он переопределяется для реализации их естественных представлений о равенстве. Это также означает, что в Scala вы никогда не попадете в хорошо известную в Java ловушку, касающуюся сравнения строк. В Scala сравнение строк работает вполне корректно:

```
scala> val x = "abcd".substring(2)
x: String = cd

scala> val y = "abcd".substring(2)
y: String = cd

scala> x == y

res12: Boolean = true
```

В Java результатом сравнения `x` с `y` будет `false`. В таком случае программисты вынуждены пользоваться методом `equals`, но эту тонкость нетрудно упустить из виду.

Может сложиться и такая ситуация, при которой вместо равенства, определенного пользователем, нужно проверить эквивалентность ссылок. Например, в ряде случаев, когда на первый план выдвигается эффективность, в отношении некоторых классов можно отдать предпочтение применению технологии *hash cons* и сравнить их экземпляры с помощью эквивалентности ссылок[83]. При этом в классе `AnyRef` определяется дополнительный метод `eq`, который не может быть переопределен

и реализован как эквивалентность ссылок (то есть для ссылочных типов он ведет себя, как == в Java). Существует также отрицание eq, которое называется ne, например:

```
scala> val x = new String("abc")
x: String = abc

scala> val y = new String("abc")
y: String = abc

scala> x == y
res13: Boolean = true


scala> x eq y
re14: Boolean = false


scala> x ne y

res15: Boolean = true
```

Более подробно равенство в Scala рассматривается в главе 30.

11.3. Низшие типы

Ниже всех в иерархии на рис. 11.1 показаны два класса: scala.Null и scala.Nothing. Это особые типы, единообразно сглаживающие острые углы имеющейся в Scala объектно-ориентированной системы типов.

Класс Null является типом нулевой ссылки: он представляет собой подкласс каждого ссылочного класса, то есть каждого класса, который сам является наследником класса AnyRef. Null несовместим с типами значений. Нельзя, к примеру, присвоить

значение null целочисленной переменной:

```
scala> val i: Int = null
<console>:7: error: an expression of type Null is
ineligible
for implicit conversion
       val i: Int = null
                    ^
```

Тип Nothing находится на самом дне иерархии классов Scala: он представляет собой подтип любого другого типа. Но значений этого типа вообще не существует. А зачем нужен тип без значений? Как говорилось в разделе 7.4, Nothing используется, в частности, чтобы сигнализировать об аварийном завершении операции.

Например, в Predef-объекте стандартной библиотеки Scala есть метод error, имеющий следующее определение:

```
def error(message: String): Nothing =
  throw new RuntimeException(message)
```

Возвращаемым типом метода error является Nothing, что говорит пользователю о ненормальном возвращении из метода (вместо этого метод выдал исключение). Поскольку Nothing — подтип любого другого типа, методы, подобные error, допускают весьма гибкое использование, например:

```
def divide(x: Int, y: Int): Int =
  if (y != 0) x / y
  else error("can't divide by zero")
```

Ветвь then данного условия, представленная выражением x / y, имеет тип Int, а ветвь else, то есть вызов error, имеет тип Nothing. Поскольку Nothing является подтипом Int, типом всего условного выражения, как и требовалось, является Int.

11.4. Определение своих собственных классов значений

В разделе 11.1 говорилось, что в дополнение к встроенным можно определять собственные классы значений. Как и экземпляры встроенных классов значений, экземпляры ваших классов значений будут, как правило, компилироваться в байт-код Java, который не использует класс-оболочку. В том контексте, где нужна оболочка, например при использовании обобщенного кода, значения будут упаковываться и распаковываться автоматически.

Классами значений можно сделать только вполне определенные классы. Чтобы класс стал классом значений, у него должен быть только один параметр и он не должен иметь внутри ничего, кроме `def`-определений. Более того, класс значений не может расширяться никакими другими классами и в классе значений не могут переопределяться методы `equals` или `hashCode`.

Чтобы определить класс значений, его нужно сделать подклассом класса `AnyVal` и поставить перед его единственным параметром префикс `val`. Пример класса значений выглядит следующим образом:

```
class Dollars(val amount: Int) extends AnyVal {
  override def toString() = "$" + amount
}
```

В соответствии с описанием, приведенным в разделе 10.6, префикс `val` позволяет иметь доступ к параметру `amount` как полю. Например, следующий код создает экземпляр класса значений, а затем извлекает из него `amount`:

```
scala> val money = new Dollars(1000000)
money: Dollars = $1000000

scala> money.amount
res16: Int = 1000000
```

В примере `money` ссылается на экземпляр класса значений. Эта переменная в исходном коде Scala имеет тип `Dollars`, но скомпилированный байт-код Java будет напрямую использовать тип `Int`.

В этом примере определяется метод `toString`, и компилятор понимает, когда его использовать. Именно поэтому вывод значения `money` дает результат `$1000000` со знаком доллара, а вывод `money.amount` дает результат `1000000`. Можно даже определить несколько типов значений, и все они будут опираться на одно и то же `Int`-значение, например:

```
class SwissFrancs(val amount: Int) extends AnyVal
{
  override def toString() = amount + " CHF"
}
```

Хотя и `Dollars`, и `SwissFrancs` представлены в виде целых чисел, ничто не препятствует их использованию в одной и той же области видимости:

```
scala> val dollars = new Dollars(1000)
dollars: Dollars = $1000

scala> val francs = new SwissFrancs(1000)
francs: SwissFrancs = 1000 CHF
```

Уход от монокультурности типов. Чтобы получить от иерархии классов Scala наибольшие преимущества, старайтесь для каждого понятия предметной области определять новый класс, несмотря на то что будет возможность повторно использовать один и тот же класс для различных целей. Даже если этот класс относится к так называемому *минимальному типу*, не имеющему методов или полей, определение дополнительного класса поможет компилятору принести вам больше пользы.

Предположим, к примеру, что вы написали некий код для создания структуры HTML. В HTML название стиля представлено в виде строки. То же самое касается и идентификаторов гипертекстовой ссылки `anchor`. Сам код HTML также является строкой, поэтому при желании для представления всего здесь перечисленного можно определить вспомогательный код, используя строки

```
def title(text: String, anchor: String, style: String): String =
  s"<a id='$anchor'><h1 class='$style'>$text</h1></a>"
```

В этой сигнатуре типа имеется четыре строки! Такой *строчно-типизированный* код с технической точки зрения является строго типизированным, но, поскольку все, что находится здесь в поле зрения, относится к типу `String`, компилятор не может помочь вам определить использование одного элемента структуры при намерении указать на применение другого элемента. Например, он не сможет уберечь вас от следующего искажения структуры:

```
scala> title("chap:vcls", "bold", "Value Classes")
res17: String = <a id='bold'><h1 class='Value
    Classes'>chap:vcls</h1></a>
```

Код HTML нарушен. При намерении вывести на экран текст `Value Classes` эта строка была использована в качестве стилевого класса, а на экран был выведен текст `chap.vcls`, для которого предполагалась роль гипертекстовой ссылки. В довершение ко всему в качестве идентификатора гипертекстовой ссылки выступила строка `bold`, для которой предполагалась роль стилевого класса. Несмотря на всю эту комедию ошибок, компилятор ничем этому не воспротивился.

Если определить для каждого понятия предметной области минимальный тип, то компилятор сможет принести больше

пользы. Например, можно определить собственный небольшой класс для стилей, идентификаторов гипертекстовых ссылок, отображаемого текста и кода HTML. Поскольку эти классы имеют один параметр и не имеют элементов, они могут быть определены как классы значений:

```
class Anchor(val value: String) extends AnyVal
class Style(val value: String) extends AnyVal
class Text(val value: String) extends AnyVal
class Html(val value: String) extends AnyVal
```

При наличии этих классов появляется возможность создать версию `title`, обладающую менее тривиальной сигнатурой наподобие такой:

```
def title(text: Text, anchor: Anchor, style:
Style): Html =
  new Html(
    s"<a id='${anchor.value}'>" +
        s"<h1 class='${style.value}'>" +
        text.value +
        "</h1></a>"
)
```

Теперь при попытке воспользоваться этой версией с указанием аргументов в неверном порядке компилятор сможет обнаружить ошибку, например:

```
scala>    title(new    Anchor("chap:vcls"),    new
Style("bold"),
           new Text("Value Classes"))
<console>:18: error: type mismatch;
Found   : Anchor
required: Text
                new Anchor("chap:vcls"),
```

```
                ^
<console>:19: error: type mismatch;
found   : Style
required: Anchor
                new Style("bold"),
                ^
<console>:20: error: type mismatch;
found   : Text
required: Style
                new Text("Value Classes"))
                ^
```

Резюме

В этой главе были показаны классы, находящиеся на самом верху и в самом низу иерархии классов Scala. Теперь, получив твердую базу знаний об иерархии классов в Scala, вы готовы к усвоению понятия смешанных композиций. Следующая глава будет посвящена трейтам.

81 Единственный случай, когда использование == не приводит к непосредственному вызову equals, относится к упакованным числовым классам Java, таким как Integer или Long. В Java new Integer(1) не эквивалентен new Long(1) даже при использовании элементарных значений 1 == 1L. Поскольку Scala — более правильный язык, чем Java, появилась необходимость скорректировать это несоответствие, применив для классов особую версию метода ==. Точно так же метод ## обеспечивает Scala-версию хеширования и похож на Java-метод hashCode, за исключением того что для упакованных числовых типов он всегда работает с методом ==. Например, для new Integer(1) и new Long(1) метод ## вырабатывает один и тот же хеш, при том что Java-методом hashCodes для них вырабатывается разный хеш-код.

82 Причина существования псевдонима AnyRef, заменяющего использование имени java.lang.Object, заключается в том, что Scala изначально разрабатывался для работы как на платформе Java, так и на платформе .NET. На платформе .NET AnyRef был псевдонимом для System.Object.

83 Hash cons-экземпляры класса создаются путем кэширования всех созданных экземпляров в слабую коллекцию. Затем, как только потребуется новый экземпляр класса, сначала проверяется кэш. Если в нем уже есть элемент, равный тому, который вы намереваетесь создать, можно повторно воспользоваться существующим экземпляром. В

результате такой систематизации любые два экземпляра, равенство которых определяется с помощью метода equals(), также равны на основе эквивалентности ссылок.

12. Трейты

Трейты в Scala являются фундаментальными многократно используемыми блоками кода. В трейте инкапсулируются определения тех методов и полей, которые затем могут многократно использоваться путем их подмешивания в классы. В отличие от наследования классов, в котором каждый класс должен быть наследником только одного родительского класса, в класс может подмешиваться любое количество трейтов. В этой главе вы узнаете, как работают трейты. Далее мы рассмотрим два наиболее распространенных способа их применения: расширение скудных интерфейсов и превращение их в насыщенные, а также определение наращиваемых изменений. Здесь будет показано, как используется трейт `Ordered`, и проведено сравнение трейтов с множественным наследованием, имеющимся в других языках.

12.1. Как работают трейты

Определение трейта похоже на определение класса, за исключением того, что в нем используется ключевое слово `trait`. Пример показан в листинге 12.1.

Листинг 12.1. Определение трейта Philosophical

```
trait Philosophical {
  def philosophize() = {
    println("I consume memory, therefore I am!")
  }
}
```

Этот трейт называется `Philosophical`. В нем не объявлен родительский класс, следовательно, как и у класса, у него есть

родительский класс по умолчанию, и это AnyRef. Здесь определяется один конкретный метод по имени philosophize. Это простой трейт, и его вполне достаточно, чтобы показать, как работают трейты.

После того как трейт определен, он может быть подмешан в класс с использованием либо ключевого слова extends, либо ключевого слова with. Программисты, работающие со Scala, подмешивают трейты, а не устраивают их наследование, поскольку подмешивание трейта имеет важное отличие от множественного наследования, встречающегося во многих других языках. Этот вопрос рассматривается в разделе 12.6. Например, в листинге 12.2 показан класс, в который с помощью ключевого слова extends подмешивается трейт Philosophical.

Листинг 12.2. Подмешивание трейта с использованием ключевого слова extends

```
class Frog extends Philosophical {
  override def toString = "green"
}
```

Для подмешивания трейта можно использовать ключевое слово extends, в таком случае происходит подразумеваемое наследование родительского класса трейта. Например, в листинге 12.2 класс Frog (лягушка) приобретает родительский класс AnyRef (это родительский класс для трейта Philosophical) и подмешивает в себя трейт Philosophical. Методы, унаследованные от трейта, могут использоваться точно так же, как и методы, унаследованные от родительского класса. Рассмотрим пример:

```
scala> val frog = new Frog
frog: Frog = green
```

```
scala> frog.philosophize()
I consume memory, therefore I am!
```

Трейт также определяет тип. Рассмотрим пример, в котором `Philosophical` используется как тип:

```
scala> val phil: Philosophical = frog
phil: Philosophical = green

scala> phil.philosophize()
I consume memory, therefore I am!
```

Типом `phil` является `Philosophical`, то есть трейт. Таким образом, переменная `phil` может быть инициализирована любым объектом, в чей класс подмешивается трейт `Philosophical`.

Если нужно подмешать трейт в класс, который расширяет родительский класс явным образом, ключевое слово `extends` применяется для указания родительского класса, а для подмешивания трейта используется ключевое слово `with`. Пример показан в листинге 12.3. Если нужно подмешать сразу несколько трейтов, дополнительные трейты указываются с помощью ключевого слова `with`. Например, располагая трейтом `HasLegs`, вы, как показано в листинге 12.4, можете подмешать в класс `Frog` как трейт `Philosophical`, так и трейт `HasLegs`.

Листинг 12.3. Подмешивание трейта с использованием ключевого слова with

```
class Animal

class Frog extends Animal with Philosophical {
  override def toString = "green"
```

```
}
```

Листинг 12.4. Подмешивание сразу нескольких трейтов

```
class Animal
trait HasLegs

class Frog extends Animal with Philosophical with
HasLegs {
  override def toString = "green"
}
```

В показанных ранее примерах класс `Frog` наследовал реализацию метода `philosophize` из трейта `Philosophical`. В качестве альтернативного варианта метод `philosophize` в классе `Frog` может быть переопределен. Синтаксис выглядит точно так же, как и при переопределении метода, объявленного в родительском классе. Рассмотрим пример:

```
class Animal

class Frog extends Animal with Philosophical {
  override def toString = "green"
  override def philosophize() = {
     println("It ain't easy being " + toString +
"!")
  }
}
```

Поскольку в новое определение класса `Frog` по-прежнему подмешивается трейт `Philosophical`, его, как и раньше, можно использовать из переменной этого типа. Но так как во `Frog` переопределено определение метода `philosophize`, которое было

дано в трейте Philosophical, при вызове будет получено новое поведение:

```
scala> val phrog: Philosophical = new Frog
phrog: Philosophical = green

scala> phrog.philosophize()
It ain't easy being green!
```

Теперь можно прийти к философскому умозаключению, что трейты подобны Java-интерфейсам с конкретными методами, но фактически их возможности гораздо шире. В трейтах могут, к примеру, объявляться поля и сохраняться состояние. Фактически в определении трейта можно делать то же самое, что и в определении класса, и синтаксис выглядит почти так же, но с двумя исключениями.

Начнем с того, что в трейте не может быть никаких присущих классу параметров (то есть параметров, передаваемых первичному конструктору класса). Иными словами, хотя у вас есть возможность определить класс таким вот образом:

```
class Point(x: Int, y: Int)
```

следующая попытка определить трейт окажется неудачной:

```
trait NoPoint(x: Int, y: Int) // Не пройдет компиляцию
```

Способ обхода этого ограничения будет показан в разделе 20.5.

Другое отличие классов от трейтов заключается в том, что в классах вызовы super имеют статическую привязку, а в трейтах — динамическую. Если в классе воспользоваться кодом super.toString, вы будете точно знать, какая именно реализация метода будет вызвана. Но когда точно такой же код применяется в трейте, то вызываемая с помощью super

реализация метода при определении трейта еще не установлена. Вызываемая реализация станет определяться заново при каждом подмешивании трейта в конкретный класс. Такое своеобразное поведение `super` является ключевым фактором, позволяющим трейтам работать в качестве наращиваемых изменений, и рассматривается в разделе 12.5. А правила разрешения `super`-вызовов будут изложены в разделе 12.6.

12.2. Сравнение скудных интерфейсов с насыщенными

Чаще всего трейты используются для автоматического добавления к классу методов в дополнение к тем методам, которые в нем уже имеются. То есть трейты способны обогатить *скудный* интерфейс, превратив его в *насыщенный*.

Противопоставление скудных интерфейсов насыщенным представляет собой компромисс, который встречается в объектно-ориентированном проектировании довольно часто. Это компромисс между теми, кто реализует интерфейс, и теми, кто им пользуется. В насыщенных интерфейсах предусмотрено множество методов, обеспечивающих удобства для тех, кто их вызывает. Клиенты могут выбрать метод, целиком отвечающий их функциональным запросам. В то же время скудный интерфейс имеет незначительное количество методов и поэтому проще обходится реализаторам. Но клиентам, обращающимся к скудным интерфейсам, приходится создавать больше собственного кода. При более скудном выборе доступных для вызова методов они могут выбирать то, что хотя бы в какой-то мере отвечает их потребностям, а для использования выбранного метода создавать дополнительный код.

Характер Java-интерфейсов чаще скудный, чем насыщенный. Например, интерфейс `CharSequence`, появившийся в Java 1.4, является скудным, общим для всех строкоподобных классов, содержащих последовательность символов. А вот как выглядит

определение этого интерфейса, если его рассматривать в качестве Scala-трейта:

```
trait CharSequence {
  def charAt(index: Int): Char
  def length: Int
     def subSequence(start: Int, end: Int): CharSequence
  def toString(): String
}
```

Хотя к любой последовательности символов, для которой предназначен интерфейс `CharSequence`, может применяться большинство из десятков методов, имеющихся в классе `String`, в Java-интерфейсе `CharSequence` объявляются всего лишь четыре метода. Если бы в `CharSequence` был включен полный `String`-интерфейс, это бы стало гораздо более весомой нагрузкой на реализаторов `CharSequence`. Каждому программисту, реализующему `CharSequence` в Java, пришлось бы определять десятки дополнительных методов. Поскольку в трейтах Scala могут содержаться конкретные методы, они делают насыщенные интерфейсы намного удобнее. Добавление в трейт конкретного метода уводит компромисс «скудный — насыщенный» в сторону более насыщенных интерфейсов. В отличие от Java, добавление конкретного метода к Scala-трейту является одноразовым действием. Вам нужно единожды реализовать тот или иной метод, сделав это в самом трейте, вместо того чтобы возиться с его повторной реализацией для каждого класса, в который подмешивается трейт. Таким образом, на создание насыщенных интерфейсов в Scala затрачивается меньше работы, чем в языках без трейтов.

Для обогащения интерфейсов с помощью трейтов нужно просто определить трейт с небольшим количеством абстрактных методов, то есть со скудной частью трейтового интерфейса, и с

потенциально большим количеством конкретных методов, реализованных применительно к абстрактным методам. Затем можно будет подмешать обогащающий трейт в класс, реализовав скудную часть интерфейса, и получить в результате класс, позволяющий обеспечить доступ ко всему насыщенному интерфейсу.

12.3. Пример: прямоугольные объекты

В графических библиотеках часто содержится множество различных классов, представляющих что-либо прямоугольное. В качестве примеров можно привести окна, растровые изображения и области, выбираемые с помощью мыши. Чтобы такие прямоугольные объекты было удобнее использовать, необходимо получать от библиотеки ответы на относящиеся к этим объектам геометрические запросы, например ширины (`width`), высоты (`height`), левого края (`left`), правого края (`right`), верхнего левого угла (`topLeft`) и т. д. Существует множество таких методов, которыми было бы неплохо располагать, следовательно, чтобы предоставить библиотеке Java все эти методы для всех прямоугольных объектов, создателям библиотеки придется выполнять непосильную работу. Для сравнения: если бы такая же библиотека была написана на Scala, ее создатели без особого труда могли бы предоставить все эти удобные методы каким угодно классам, воспользовавшись трейтами.

Чтобы понять, как это делается, представим сначала, как бы все это могло выглядеть без использования трейтов. Должны существовать какие-то основные геометрические классы вроде точки — `Point` и прямоугольника — `Rectangle`:

```
class Point(val x: Int, val y: Int)

class Rectangle(val topLeft: Point, val
bottomRight: Point) {
```

```
  def left = topLeft.x
  def right = bottomRight.x
  def width = right - left
  // и множество других геометрических методов...
}
```

Класс `Rectangle` получает в свой первичный конструктор две точки: координаты верхнего левого и нижнего правого углов. Затем путем простых вычислений, основанных на координатах этих двух точек, он реализует множество удобных методов, таких как `left`, `right` и `width`.

Еще один класс, который может войти в состав графической библиотеки, касается двумерного графического виджета:

```
abstract class Component {
  def topLeft: Point
  def bottomRight: Point

  def left = topLeft.x
  def right = bottomRight.x
  def width = right - left
  // и множество других геометрических методов...
}
```

Обратите внимание на то, что определения `left`, `right` и `width` в обоих классах абсолютно одинаковы. Практически одинаковыми, с незначительными вариациями, они будут и в любых других классах, создаваемых для прямоугольных объектов.

Уменьшить эту повторяемость можно с помощью обогащающего трейта. Он будет иметь два абстрактных метода: один возвращает верхнюю левую координату объекта, а другой — нижнюю правую. Затем в нем могут быть предоставлены конкретные реализации решения всех других геометрических запросов. На что это будет похоже, показано в листинге 12.5.

Листинг 12.5. Определение обогащающего трейта

```
trait Rectangular {
  def topLeft: Point
  def bottomRight: Point

  def left = topLeft.x
  def right = bottomRight.x
  def width = right - left
  // и множество других геометрических методов...
}
```

Этот трейт можно подмешать в класс Component, чтобы получить все геометрические методы, предоставляемые Rectangular:

```
abstract class Component extends Rectangular {
  // другие методы...
}
```

Точно так же трейт может подмешиваться в сам Rectangle:

```
class Rectangle(val topLeft: Point, val
bottomRight: Point)
  extends Rectangular {

  // другие методы...
}
```

Располагая этими определениями, можно создать Rectangle-объект и вызвать в отношении него геометрические методы width и left:

```
scala> val rect = new Rectangle(new Point(1, 1),
           new Point(10, 10))
```

```
rect: Rectangle = Rectangle@5f5da68c

scala> rect.left
res2: Int = 1

scala> rect.right
res3: Int = 10

scala> rect.width
res4: Int = 9
```

12.4. Трейт Ordered

Сравнение является еще одной областью, где может пригодиться насыщенный интерфейс. При сравнении двух упорядоченных объектов было бы удобно воспользоваться вызовом одного метода, чтобы выяснить результаты желаемого сравнения. Если нужно использовать сравнение «меньше чем», предпочтительнее было бы вызвать <, а если нужно использовать сравнение «меньше чем или равно» — вызвать <=. Со скудным интерфейсом можно располагать только методом <, и тогда временами приходилось бы создавать код вроде `(x < y) || (x == y)`. Насыщенный интерфейс предоставит вам все привычные операторы сравнения, позволяя напрямую создавать код вроде `x <= y`.

Перед тем как посмотреть на трейт `Ordered`, представим себе, что можно сделать без него. Предположим, вы взяли класс `Rational` из главы 6 и добавили к нему операции сравнения. У вас должен получиться примерно такой код[84]:

```
class Rational(n: Int, d: Int) {
  // ...
  def < (that: Rational) =
        this.numer  *  that.denom  <  that.numer  *
```

```
this.denom
  def > (that: Rational) = that < this
  def <= (that: Rational) = (this < that) || (this
== that)
  def >= (that: Rational) = (this > that) || (this
== that)
}
```

В этом классе определяются четыре оператора сравнения (<, >, <= и >=), и это классическая демонстрация стоимости определения насыщенного интерфейса. Сначала обратите внимание на то, что три оператора сравнения созданы на основе первого оператора. Например, оператор > определен как противоположность оператора <, а оператор <= — буквально как «меньше чем или равно». Затем обратите внимание на то, что все три метода будут такими же для любого другого класса, объекты которого могут сравниваться друг с другом. В отношении оператора <= для рациональных чисел не прослеживается никаких особенностей. В контексте сравнения оператор <= всегда используется для обозначения «меньше чем или равно». В общем, в этом классе есть довольно много шаблонного кода, который будет точно таким же в любом другом классе, реализующем операции сравнения.

Такая проблема встречается настолько часто, что в Scala предоставляется трейт, помогающий справиться с ее решением. Этот трейт называется `Ordered`. Чтобы им воспользоваться, нужно заменить все отдельные методы сравнений одним методом `compare`. Затем на основе одного этого метода определить в трейте `Ordered` методы <, >, <= и >=. Таким образом, трейт `Ordered` позволит вам обогатить класс методами сравнений путем реализации всего одного метода по имени `compare`.

Если определить операции сравнений в `Rational` путем использования трейта `Ordered`, то код будет иметь следующий вид:

```
class Rational(n: Int, d: Int) extends
Ordered[Rational] {
  // ...
  def compare(that: Rational) =
      (this.numer * that.denom) - (that.numer *
this.denom)
}
```

Нужно выполнить две задачи. Начнем с того, что в `Rational` подмешивается трейт `Ordered`. В отличие от трейтов, которые встречались до сих пор, `Ordered` требует от вас при подмешивании указать параметр типа. Параметры типов до главы 19 подробно рассматриваться не будут, а пока все, что нужно знать при подмешивании `Ordered`, сводится к тому, что фактически нужно выполнять подмешивание `Ordered[C]`, где C обозначает класс, элементы которого сравниваются. В данном случае в `Rational` подмешивается `Ordered[Rational]`.

Вторая задача, требующая выполнения, заключается в определении метода `compare` для сравнения двух объектов. Этот метод должен сравнивать получатель `this` с объектом, переданным методу в качестве аргумента. Возвращать он должен целочисленное значение, которое равно нулю, если объекты одинаковы, отрицательное число, если получатель меньше аргумента, и положительное число, если получатель больше аргумента.

В данном случае метод сравнения класса `Rational` использует формулу, основанную на приведении чисел к общему знаменателю с последующим вычитанием получившихся числителей. Теперь при наличии этого подмешивания и определения метода `compare` класс `Rational` имеет все четыре метода сравнения:

```
scala> val half = new Rational(1, 2)
half: Rational = 1/2
```

```
scala> val third = new Rational(1, 3)
third: Rational = 1/3

scala> half < third
res5: Boolean = false

scala> half > third
res6: Boolean = true
```

При каждой реализации класса с какой-либо возможностью упорядоченности путем сравнения нужно рассматривать вариант подмешивания в него трейта `Ordered`. Если выполнить это подмешивание, пользователи класса получат богатый набор методов сравнений.

Имейте в виду, что в трейте `Ordered` критерий равенства за вас не определяется, поскольку трейт на это не способен. Дело в том, что реализация установления критерия равенства применительно к сравнению требует проверки типа переданного объекта, а по причине затирания типов сам `Ordered` эту проверку выполнить не может. Поэтому определять критерий равенства вам придется самим, даже если вы подмешиваете трейт `Ordered`. Как справиться с этой задачей, будет объяснено в главе 30.

Целиком трейт `Ordered` за вычетом комментариев и средств обеспечения совместимости выглядит следующим образом:

```
trait Ordered[T] {
  def compare(that: T): Int

  def <(that: T): Boolean = (this compare that) <
0
  def >(that: T): Boolean = (this compare that) >
0
  def <=(that: T): Boolean = (this compare that)
<= 0
```

```
  def >=(that: T): Boolean = (this compare that)
>= 0
}
```

Насчет обозначений `T` и `[T]` волноваться не стоит. Здесь `T` обозначает параметр типа, который подробнее рассматривается в главе 19. Чтобы разобраться с трейтом `Ordered`, просто думайте о параметре типа как о типе, совпадающем с типом получателя. Затем можно увидеть, что в этом трейте определяется один абстрактный метод `compare`, от которого ожидается сравнение получателя (`this`) с другим объектом (`that`) такого же типа, что и получатель. Располагая одним этим методом, `Ordered` может затем предоставить конкретные определения методов <, >, <= и >=.

12.5. Трейты в качестве наращиваемых изменений

Основное применение трейтов — превращение скудного интерфейса в насыщенный — вы уже видели. Перейдем теперь ко второму по значимости способу применения трейтов — предоставлению классам наращиваемых изменений. Трейты дают возможность изменять методы класса, позволяя вам накладывать их друг на друга.

Рассмотрим в качестве примера наращиваемые изменения применительно к очереди целых чисел. В ней будут две операции: `put`, помещающая целые числа в очередь, и `get`, извлекающая их из очереди. Очередь работает по принципу «первым пришел, первым ушел», поэтому возвращение целых чисел из нее происходит в том же порядке, в котором они туда были помещены.

Располагая классом, реализующим такую очередь, можно определять трейты для выполнения следующих изменений:

- удваивания — удваиваются все целые числа, помещенные в очередь;

- увеличения на единицу — увеличиваются все целые числа, помещенные в очередь;
- фильтрации — из очереди отфильтровываются все отрицательные целые числа.

Эти три трейта представляют модификации, поскольку изменяют поведение соответствующего класса очереди, не давая полного определения самому классу очереди. Все три трейта также являются *наращиваемыми*. Можно выбрать любые из трех трейтов, подмешать их к классу и получить новый класс, обладающий всеми выбранными модификациями.

В листинге 12.6 представлен абстрактный класс `IntQueue`. В `IntQueue` имеются метод `put`, добавляющий к очереди новые целые числа, и метод `get`, возвращающий целые числа и удаляющий их из очереди. Основная реализация `IntQueue`, использующая `ArrayBuffer`, показана в листинге 12.7.

Листинг 12.6. Абстрактный класс IntQueue

```
abstract class IntQueue {
  def get(): Int
  def put(x: Int)
}
```

Листинг 12.7. Реализация класса BasicIntQueue с использованием ArrayBuffer

```
import scala.collection.mutable.ArrayBuffer

class BasicIntQueue extends IntQueue {
  private val buf = new ArrayBuffer[Int]
```

```
  def get() = buf.remove(0)
  def put(x: Int) = { buf += x }
}
```

В классе BasicIntQueue имеется закрытое поле, содержащее буфер в виде массива. Метод get удаляет запись с одного конца буфера, а метод put добавляет элементы к другому его концу. Пример использования данной реализации выглядит следующим образом:

```
scala> val queue = new BasicIntQueue
queue: BasicIntQueue = BasicIntQueue@23164256

scala> queue.put(10)

scala> queue.put(20)

scala> queue.get()
res9: Int = 10

scala> queue.get()
res10: Int = 20
```

Пока все вроде бы в порядке. Теперь посмотрим на использование трейтов для модификации этого поведения. В листинге 12.8 показан трейт, удваивающий целые числа по мере их помещения в очередь. Трейт Doubling имеет две интересные особенности. Первая заключается в том, что в нем объявляется родительский класс IntQueue. Это означает, что трейт может подмешиваться только в класс, который также расширяет IntQueue. То есть Doubling можно подмешивать в BasicIntQueue, но не в Rational.

Листинг 12.8. Трейт наращиваемых изменений Doubling

```
trait Doubling extends IntQueue {
    abstract override def put(x: Int) = { super.put(2 * x) }
}
```

Вторая интересная особенность заключается в том, что у трейта имеется вызов `super` в отношении метода, объявленного абстрактным. Для обычных классов такие вызовы применять запрещено, поскольку на этапе выполнения они гарантированно дадут сбой. Но для трейта такой вызов может действительно пройти успешно. Поскольку в трейте `super`-вызовы имеют динамическую привязку, `super`-вызов в трейте `Doubling` будет работать при условии, что трейт подмешан в другой трейт или класс, в котором дается конкретное определение метода.

Трейтам, реализующим наращиваемые изменения, зачастую нужен именно такой порядок. Чтобы сообщить компилятору, что это делается намеренно, такие методы следует помечать модификаторами `abstract override`. Это сочетание модификаторов позволительно только для элементов трейтов, но не классов, и означает, что трейт должен быть подмешан в некий класс, имеющий конкретное определение рассматриваемого метода.

Вроде бы простой трейт, а сколько вокруг него складывается различных обстоятельств! Применение трейта выглядит следующим образом:

```
scala> class MyQueue extends BasicIntQueue with Doubling
defined class MyQueue

scala> val queue = new MyQueue
queue: MyQueue = MyQueue@44bbf788
```

```
scala> queue.put(10)

scala> queue.get()
res12: Int = 20
```

В первой строке этого сеанса работы с интерпретатором определяется класс MyQueue, расширяющий класс BasicIntQueueand путем подмешивания в него трейта Doubling. Мы помещаем в очередь число 10, но в результате подмешивания трейта Doubling число 10 удваивается. При извлечении целого числа из очереди оно уже имеет значение 20.

Обратите внимание на то, что в MyQueue не определяется никакого нового кода — просто идентифицируется класс и подмешивается трейт. В такой ситуации вместо задания именованного класса код BasicIntQueue with Doubling может быть предоставлен непосредственно с ключевым словом new. Работа такого кода показана в листинге 12.9.

Листинг 12.9. Подмешивание трейта при создании экземпляра с помощью ключевого слова new

```
scala> val queue = new BasicIntQueue with Doubling
queue: BasicIntQueue with Doubling = $anon$1@141f05bf

scala> queue.put(10)

scala> queue.get()
res14: Int = 20
```

Чтобы посмотреть на наращивание изменений, нужно определить еще два модифицирующих трейта, Incrementing и

Filtering. Реализация этих трейтов показана в листинге 12.10.

Листинг 12.10. Трейты наращиваемых изменений Incrementing и Filtering

```
trait Incrementing extends IntQueue {
    abstract override def put(x: Int) = { super.put(x + 1) }
}
trait Filtering extends IntQueue {
  abstract override def put(x: Int) = {
    if (x >= 0) super.put(x)
  }
}
```

Теперь, располагая модифицирующими трейтами, можно выбрать, какой из них вам понадобится для той или иной очереди. Например, здесь показана очередь, в которой не только отфильтровываются отрицательные числа, но и ко всем сохраняемым числам прибавляется единица:

```
scala> val queue = (new BasicIntQueue
           with Incrementing with Filtering)
queue: BasicIntQueue with Incrementing with Filtering...

scala> queue.put(-1); queue.put(0); queue.put(1)

scala> queue.get()
res16: Int = 1

scala> queue.get()
res17: Int = 2
```

Порядок подмешиваний играет существенную роль[85]. Конкретные правила даны в следующем разделе, но, грубо говоря, тот трейт, что находится правее, вступает в силу первым. Когда метод вызывается в отношении экземпляра класса с подмешанными трейтами, то первым вызывается тот метод, который определен в самом правом трейте. Если этот метод выполняет `super`-вызов, то вызывается метод, который определен в следующем трейте левее данного трейта, и т. д. В предыдущем примере сначала вызывается метод `put` трейта `Filtering`, следовательно, все начинается с того, что он удаляет отрицательные целые числа. Вторым вызывается метод `put` трейта `Incrementing`, следовательно, к оставшимся целым числам прибавляется единица.

Если расположить трейты в обратном порядке, то сначала к целым числам будет прибавляться единица и только потом те целые числа, которые все же останутся отрицательными, будут удалены:

```
scala> val queue = (new BasicIntQueue
           with Filtering with Incrementing)
queue: BasicIntQueue with Filtering with
Incrementing...

scala> queue.put(-1); queue.put(0); queue.put(1)

scala> queue.get()
res19: Int = 0

scala> queue.get()
res20: Int = 1

scala> queue.get()
res21: Int = 2
```

В общем, код, создаваемый в данном стиле, открывает перед вами широкие возможности для проявления гибкости. Подмешивая эти три трейта в разных сочетаниях и разном порядке следования, можно определить 16 различных классов. Весьма впечатляющая гибкость для столь незначительного объема кода, поэтому постарайтесь не проглядеть возможности организации кода для получения наращиваемых изменений.

12.6. Почему не используется множественное наследование

Трейты являются способом наследования из множества похожих на классы структурных компонентов, но они имеют весьма важные отличия от множественного наследования, имеющегося во многих языках программирования. Одно из отличий, интерпретация `super`, играет особенно важную роль. При множественном наследовании метод, вызванный с помощью `super`-вызова, может быть определен прямо там, где появляется этот вызов. При использовании трейтов вызываемый метод определяется путем *линеаризации*, то есть выстраивания в ряд классов и трейтов, подмешанных в класс. Это то самое рассмотренное в предыдущем разделе отличие, которое позволяет выполнять наращивание изменений.

Перед тем как рассмотреть линеаризацию, немного отвлечемся на то, как наращиваемые изменения выполняются в языке с традиционным множественным наследованием. Представим себе следующий код, но на этот раз интерпретируемый не как подмешивание трейтов, а как множественное наследование:

```
// Мысленный эксперимент с множественным наследованием
val q = new BasicIntQueue with Incrementing with Doubling
q.put(42) // Который из методов put будет вызван?
```

Сразу возникает вопрос: который из методов `put` будет

задействован в этом вызове? Возможно, вступят в силу правила, согласно которым победу одержит самый последний родительский класс. В таком случае будет вызван метод из `Doubling`. В этом методе будет удвоен его аргумент, сделан вызов `super.put`, и на этом все. Не произойдет никакого увеличения на единицу! Также, если бы действовало правило, при котором побеждал бы первый родительский класс, в получающейся очереди целые числа увеличивались бы на единицу, но не удваивались. То есть не сработало бы никакое упорядочение.

Можно также подумать о предоставлении программистам возможности точно указывать при использовании `super`-вызова, из какого именно родительского класса им нужен метод. Представим, к примеру, что есть следующий Scala-подобный код, в котором в `super` фигурирует точное указание на то, что метод вызывается как из `Incrementing`, так и из `Doubling`:

```
// Мысленный эксперимент с множественным наследованием
trait MyQueue extends BasicIntQueue
    with Incrementing with Doubling {

  def put(x: Int) = {
    Incrementing.super.put(x) // (на самом деле это не Scala)
    Doubling.super.put(x)
  }
}
```

Такой подход породит новые проблемы, самой малой из которых станет многословие. Получится, что метод `put` основного класса будет вызван дважды — один раз со значением, увеличенным на единицу, и один раз с удвоенным значением, но никогда не будет вызван с увеличенным на единицу и удвоенным значением.

При использовании множественного наследования подходящего решения этой задачи просто не существует. Придется опять возвращаться к проектированию и реорганизовывать код. В отличие от этого, с решением на основе применения трейтов в Scala все предельно понятно. Вы просто подмешиваете трейты `Incrementing` и `Doubling`, и имеющееся в Scala особое правило, касающееся применения `super` в трейтах, позволяет добиться всего, чего вы хотели. Понятно, что-то здесь отличается от традиционного множественного наследования, но что именно? Согласно уже данной подсказке ответом будет линеаризация. Когда с помощью ключевого слова `new` создается экземпляр класса, Scala берет класс со всеми его унаследованными классами и трейтами и располагает их в едином линейном порядке. Затем при любом `super`-вызове внутри одного из таких классов вызываемым становится тот метод, который идет следующим вверх по цепочке. Если во всех методах, кроме последнего, присутствует `super`-вызов, получается наращивание.

Описание конкретного порядка линеаризации дается в спецификации языка. Он сложноват, но главное, что вам нужно знать, заключается в том, что при любой линеаризации класс всегда следует впереди своих родительских классов и подмешанных трейтов. Таким образом, при написании метода, содержащего `super`-вызов, этот метод изменяет поведение родительского класса и подмешанных трейтов, а не наоборот.

ПРИМЕЧАНИЕ

Далее в этом разделе дается описание подробностей линеаризации. Вы можете пропустить этот материал, если сейчас он вам неинтересен.

Главные свойства выполняющейся в Scala линеаризации показаны в следующем примере. Предположим, что у вас есть класс `Cat` (Кот), являющийся наследником родительского класса

Animal (Животное), и два родительских трейта, Furry (Пушистый) и FourLegged (Четырехлапый). Трейт FourLegged расширяет и еще один трейт — HasLegs (С лапами):

```
class Animal
trait Furry extends Animal
trait HasLegs extends Animal
trait FourLegged extends HasLegs
class Cat extends Animal with Furry with
FourLegged
```

Иерархия наследования и линеаризация класса Cat показана на рис. 12.1. Наследование изображено с использованием традиционной системы записи UML[86]: стрелки, на концах которых белые треугольники, служат признаками наследования, упираясь в родительский тип. Темные стрелки показывают линеаризацию. Ими указывается направление, в котором будут разрешаться super-вызовы.

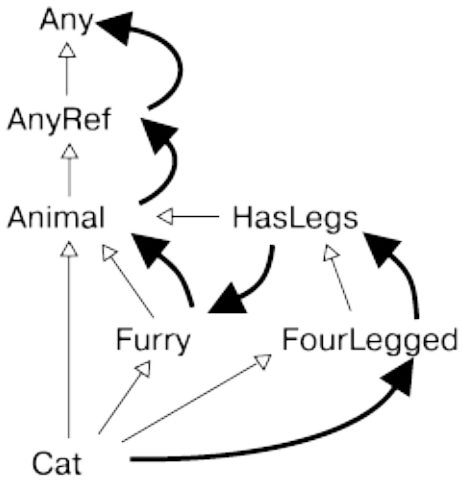


Рис. 12.1. Иерархия наследования и линеаризации класса Cat

Линеаризация Cat вычисляется с конца к началу следующим образом. Последней частью линеаризации Cat является линеаризация его родительского класса Animal. Эта линеаризация

копируется без каких-либо изменений. (Линеаризация каждого из этих типов показана в табл. 12.1.) Поскольку класс `Animal` не расширяет явным образом какой-либо родительский класс и не подмешивает никаких родительских трейтов, то по умолчанию он расширяет класс `AnyRef`, который расширяет класс `Any`. Поэтому линеаризация класса `Animal` имеет следующий вид:

`Animal` → `AnyRef` → `Any`.

Второй частью является линеаризация первой примеси, трейта `Furry`, но все классы, которые уже присутствуют в линеаризации класса `Animal`, теперь не учитываются, поэтому каждый класс в линеаризации `Cat` появляется только один раз. Результат выглядит следующим образом:

`Furry` → `Animal` → `AnyRef` → `Any`.

Всему этому предшествует линеаризация `FourLegged`, в которой также не учитываются любые классы, которые уже были скопированы в линеаризациях родительского класса или первого подмешанного трейта:

`FourLegged` → `HasLegs` → `Furry` → `Animal` → `AnyRef` → `Any`.

И наконец, первым в линеаризации класса `Cat` фигурирует сам этот класс:

`Cat` → `FourLegged` → `HasLegs` → `Furry` → `Animal` → `AnyRef` → `Any`.

Когда любой из этих классов и трейтов вызывает метод через `super`-вызов, вызываться будет первая реализация, которая в линеаризации расположена справа от него.

Таблица 12.1. Линеаризация типов в иерархии класса Cat

Тип	Линеаризация
Animal	Animal, AnyRef, Any
Furry	Furry, Animal, AnyRef, Any
FourLegged	FourLegged, HasLegs, Animal, AnyRef, Any
HasLegs	HasLegs, Animal, AnyRef, Any
Cat	Cat, FourLegged, HasLegs, Furry, Animal, AnyRef, Any

12.7. Так все же с трейтами или без?

При реализации многократно используемых поведенческих коллекций придется решать, чем воспользоваться, трейтом или абстрактным классом. Золотого правила не существует, но в этом разделе содержится несколько полезных рекомендаций, к которым стоит прислушаться.

- **Если поведение не предназначено для многократного использования**, тогда создавайте конкретный класс, поскольку в нем вообще нет повторно используемого поведения.

- **Если поведение может быть использовано многократно независимо от классов**, создайте трейт. Только трейты могут подмешиваться в различные части иерархии классов.

- **Если нужно получить наследство от класса в коде Java**, воспользуйтесь абстрактным классом. Поскольку в Java не существует близкого аналога трейтам с кодом, унаследовать что-либо из трейта в классе Java вряд ли получится. А вот наследование из класса Scala ничем не отличается от наследования из класса Java. В качестве единственного исключения следует отметить, что трейт Scala, имеющий исключительно абстрактные элементы, преобразуется непосредственно в интерфейс Java, поэтому можно спокойно определять такие трейты, даже при том что наследование из них предполагается использовать в коде Java. Более подробную

информацию о совместной работе кода на Java с кодом на Scala можно найти в главе 31.

- **Если планируется распространение программы в скомпилированном виде** и ожидается, что сторонние группы разработчиков станут создавать классы, что-либо из нее наследующие, следует отдать предпочтение использованию абстрактных классов. Проблема в том, что при появлении в трейте нового элемента или удалении из него старого все классы, являющиеся его наследниками, должны быть перекомпилированы, даже если в них не было никаких изменений. Если сторонние клиенты будут только вызывать поведение, а не наследовать его, то с использованием трейта не будет никаких проблем.

- **Если вы не пришли к решению** даже после того, как взвесили все ранее предоставленные рекомендации, тогда начните с создания трейта. Позже вы всегда сможете его изменить, а в целом использование трейта оставляет возможность применения большинства вариантов.

Резюме

В этой главе были показаны работа трейтов и порядок их использования в нескольких часто встречающихся идиомах. Вы увидели, что трейты похожи на множественное наследование. Но благодаря тому что в трейтах `super`-вызовы интерпретируются с помощью линеаризации, удается не только избавиться от некоторых трудностей традиционного множественного наследования, но и воспользоваться наращиванием изменений в поведении программы. Также был рассмотрен трейт `Ordered` и изучен порядок создания собственных обогащающих трейтов.

А теперь, усвоив все эти аспекты, нам предстоит вернуться немного назад и посмотреть на трейты в целом с другого ракурса.

Трейты не просто поддерживают средства выражения, рассмотренные в данной главе, они являются основополагающими блоками кода, допускающими их многократное использование с помощью механизма наследования. Благодаря этому многие опытные программисты, работающие на Scala, на ранних стадиях реализации начинают с трейтов. Любой трейт не может охватить всю концепцию, ограничиваясь лишь ее фрагментом. По мере того как конструкция приобретает все более четкие очертания, фрагменты путем подмешивания трейтов могут объединяться в более полноценные концепции.

84 Этот пример основан на использовании класса Rational, показанного в листинге 6.5, с эквивалентностью, хеш-кодом и изменениями, гарантирующими прибавление denom с положительным значением.

85 Поскольку трейт подмешивается в класс, его можно называть также примесью.

86 Rumbaugh J., Jacobson I. and Booch G. The Unified Modeling Language Reference Manual. — 2nd Ed. — Addison-Wesley, 2004.

13. Пакеты и импортируемый код

В ходе работы над программой, особенно объемной, важно свести к минимуму сцепление, то есть степень взаимозависимости различных частей программы. Низкая степень сцепления сокращает риск того, что незначительное, казалось бы, безобидное изменение в одной части программы повлечет за собой разрушительные последствия для другой. Одним из способов сведения сцепления к минимуму является написание программы в модульном стиле. Программа разбивается на несколько меньших по размеру модулей, у каждого из которых есть внутренняя и внешняя стороны. При работе над внутренней частью модуля, то есть над его реализацией, нужно согласовывать действия только с другими программистами, работающими над созданием того же самого модуля. И лишь при необходимости внесения изменений в его внешнюю сторону, то есть в интерфейс, приходится координировать свои действия с разработчиками других модулей.

В этой главе будут показаны конструкции, помогающие программировать в модульном стиле. В ней рассматриваются вопросы помещения кода в пакеты, создания имен, видимых при импортировании, и управления видимостью определений посредством модификаторов доступа. Эти конструкции сходны по духу с конструкциями, имеющимися в Java, за исключением некоторых различий, выражающихся, как правило, в их более последовательном характере, поэтому данную главу стоит прочитать даже тем, кто хорошо разбирается в Java.

13.1. Помещение кода в пакеты

Код Scala размещается в глобальной иерархии пакетов Java-платформы. Все показанные до сих пор в этой книге примеры кода размещались в безымянных пакетах. Поместить код в именованные пакеты в Scala можно двумя способами. Первый, как

показано в листинге 13.1, предусматривает помещение содержимого всего файла в пакет путем указания директивы `package` в самом начале файла.

Листинг 13.1. Помещение в пакет всего содержимого файла

```
package bobsrockets.navigation
class Navigator
```

Указание директивы `package` в листинге 13.1 приводит к тому, что класс `Navigator` помещается в пакет по имени `bobsrockets.navigation`. По-видимому, это программные средства навигации, разработанные корпорацией Bob Rockets.

ПРИМЕЧАНИЕ

Поскольку код Scala является частью экосистемы Java, тем пакетам Scala, которые выпускаются открытыми, рекомендуется соответствовать принятому в Java соглашению по присваиванию имени с обратным порядком следования доменных имен. Поэтому наиболее подходящим именем пакета для класса Navigator может быть com.bobsrockets.navigation. Но в данной главе мы отбросим «com», чтобы в примерах было легче разобраться.

Другой способ, позволяющий в Scala помещать код в пакеты, больше похож на использование пространств имен C#. За директивой `package` следует раздел в фигурных скобках, в котором содержатся определения, помещаемые в пакет. Такой синтаксис называется *пакетированием*. Пакетирование, представленное в листинге 13.2, имеет тот же эффект, что и код в листинге 13.1.

Листинг 13.2. Длинная форма простого объявления пакетирования

```
package bobsrockets.navigation {
  class Navigator
}
```

Для таких простых примеров можно воспользоваться также синтаксическим аналогом, показанным в листинге 13.1. Но лучше все же реализовать один из вариантов более универсальной системы записи, позволяющей разместить разные части файла в разных пакетах. Например, в тот же самый файл в качестве исходного кода можно включить тестирование классов, но, как показано в листинге 13.3, поместить тесты в другой пакет.

Листинг 13.3. Несколько пакетов в одном и том же файле

```
package bobsrockets {
  package navigation {

    // В пакете bobsrockets.navigation
    class Navigator

    package tests {

      // В пакете bobsrockets.navigation.tests
      class NavigatorSuite
    }
  }
}
```

13.2. Краткая форма доступа к родственному коду

Деление кода на иерархию пакетов не только помогает

программистам просматривать код, но и позволяет сообщить компилятору, что код в одном и том же пакете является родственным. В Scala эта родственность дает возможность при доступе к коду, находящемуся в одном и том же пакете, применять краткие имена без указания всех составляющих.

В листинге 13.4 приведены три примера. В первом согласно ожиданиям к классу можно обращаться из его собственного пакета, не указывая префикс. Именно поэтому `new StarMap` проходит компиляцию. Класс `StarMap` находится в том же самом пакете по имени `bobsrockets.navigation`, что и выражение `new`, которое к нему обращается, поэтому указывать префикс в виде имени пакета не нужно.

Листинг 13.4. Краткая форма обращения к классам и пакетам

```
package bobsrockets {
  package navigation {
    class Navigator {
       // Указывать bobsrockets.navigation.StarMap
не нужно
      val map = new StarMap
    }
    class StarMap
  }
  class Ship {
     // Указывать bobsrockets.navigation.Navigator
не нужно
    val nav = new navigation.Navigator
  }
  package fleets {
    class Fleet {
      // Указывать bobsrockets.Ship не нужно
```

```
      def addShip() = { new Ship }
    }
  }
}
```

Во втором примере обращение к самому пакету может производиться из того же пакета, в котором он находится, без указания префикса. Посмотрите в листинге 13.4, как создается экземпляр класса `Navigator`. Выражение `new` появляется в пакете `bobsrockets`, который содержится в пакете `bobsrockets.navigation`. Поэтому обращение к пакету `bobsrockets.navigation` можно указывать просто как `navigation`.

В третьем примере показано, что при использовании синтаксиса пакетирования с применением фигурных скобок все имена, доступные в пространстве имен вне пакета, доступны также и внутри него. Это обстоятельство позволяет в листинге 13.4 в `addShip()` создать новый экземпляр класса, воспользовавшись выражением `new Ship`. Метод определен внутри двух пакетов: внешнего `bobsrockets` и внутреннего `bobsrockets.fleets`. Поскольку к `Ship`-объекту доступ можно получить во внешнем пакете, из `addShip()` можно воспользоваться ссылкой на него.

Следует заметить, что такая форма доступа применима только в том случае, если пакеты вложены друг в друга явным образом. Если в каждый файл будет помещаться только один пакет, тогда, как и в Java, доступны будут только те имена, которые определены в текущем пакете. В листинге 13.5 пакет `bobsrockets.fleets` был перемещен на самый верх. Поскольку он больше не заключен в пакет `bobsrockets`, имена из `bobsrockets` в его пространстве имен отсутствуют. В результате использование выражения `new Ship` вызовет ошибку компиляции.

Листинг 13.5. Обозначения, заключенные в пакеты, не являются

автоматически доступными

```
package bobsrockets {
  class Ship
}
package bobsrockets.fleets {
  class Fleet {
    // Не пройдет компиляцию!
    // Ship не находится в области видимости.
    def addShip() = { new Ship }
  }
}
```

Если обозначение вложенности пакетов с использованием фигурных скобок приводит к неудобному для вас сдвигу кода вправо, можно воспользоваться несколькими указаниями директивы `package` без фигурных скобок[87]. Например, в показанном далее коде класс `Fleet` определяется в двух вложенных пакетах `bobrockets` и `fleets` точно так же, как это было сделано в листинге 13.4:

```
package bobsrockets
package fleets
class Fleet {
  // Указывать bobsrockets.Ship не нужно
  def addShip() = { new Ship }
}
```

Важно знать еще об одной, последней особенности. Иногда при создании программного кода получается переполненное пространство имен, в котором имена пакетов скрываются друг за другом. В листинге 13.6 пространство имен класса `MissionControl` включает три отдельных пакета по имени `launch`! Один пакет `launch` находится в

bobsrockets.navigation, один — в bobsrockets и еще один — на верхнем уровне. А как тогда ссылаться на Booster1, Booster2 и Booster3?

Листинг 13.6. Доступ к скрытым именам пакетов

```
// В файле launch.scala
package launch {
  class Booster3
}
// В файле bobsrockets.scala
package bobsrockets {
  package navigation {
    package launch {
      class Booster1
    }
    class MissionControl {
      val booster1 = new launch.Booster1
                          val    booster2    =    new
bobsrockets.launch.Booster2
      val booster3 = new _root_.launch.Booster3
    }
  }
  package launch {
    class Booster2
  }
}
```

Проще всего обратиться к первому из них. Ссылка на само имя launch приведет вас к пакету bobsrockets.navigation.launch, поскольку этот пакет launch определен в ближайшей видимости. Поэтому к первому из booster-классов можно обратиться просто

как к launch.Booster1. Ссылка на второй подобный класс также указывается без каких-либо особенных приемов. Можно указать bobrockets.launch.Booster2 и не оставить ни малейших сомнений о том, к какому из трех классов происходит обращение. Открытым остается лишь вопрос по поводу третьего booster-класса: как обратиться к Booster3 при условии, что пакет на самом верхнем уровне затеняется вложенными пакетами?

Чтобы помочь справиться с подобной ситуацией, в Scala предоставляется имя пакета _root_, являющегося внешним по отношению к любым другим создаваемым пользователем пакетам. Иначе говоря, каждый пакет верхнего уровня, который может быть создан, рассматривается как составная часть _root_. Например, и launch, и bobsrockets в листинге 13.6 являются составными частями пакета _root_. В результате этого _root_.launch позволяет обратиться к пакету launch самого верхнего уровня, а _root_.launch.Booster3 обозначает внешний booster-класс.

13.3. Импортирование кода

В Scala пакеты и их составляющие могут импортироваться с использованием директивы import. Затем ко всему, что было импортировано, можно получить доступ путем указания простого имени наподобие File, без использования такого развернутого имени, как java.io.File. Рассмотрим, к примеру, код, показанный в листинге 13.7.

Листинг 13.7. Превосходные фрукты от Боба, готовые к импорту

```
package bobsdelights

abstract class Fruit(
  val name: String,
```

```
  val color: String
)

object Fruits {
  object Apple extends Fruit("apple", "red")
  object Orange extends Fruit("orange", "orange")
  object Pear extends Fruit("pear", "yellowish")
  val menu = List(Apple, Orange, Pear)
}
```

Указание директивы `import` делает составляющие пакета или объект доступными по их именам, исключая необходимость ставить перед ними префикс с именем пакета или объекта. Рассмотрим ряд простых примеров:

```
// Простая форма доступа к Fruit
import bobsdelights.Fruit

// Простая форма доступа ко всем составляющим bobsdelights
import bobsdelights._

// Простая форма доступа ко всем составляющим Fruits
import bobsdelights.Fruits._
```

Первый пример относится к импортированию отдельно взятого Java-типа, а во втором показан Java-импорт *до востребования* (on-demand). Единственное отличие состоит в том, что импорт до востребования в Scala записывается с замыкающим знаком подчеркивания (_) вместо знака звездочки (*). (Поскольку знак * является в Scala допустимым идентификатором!) Третья из показанных директив `import` относится к Java-импорту статических полей класса.

Эти три директивы `import` дают представление о том, что можно делать с помощью импортирования, но в Scala импортирование носит более универсальный характер. В частности, импортирование в Scala может быть где угодно, а не только в начале компилируемого модуля. К тому же при импортировании можно ссылаться на произвольные значения. Например, возможен импорт, показанный в листинге 13.8.

Листинг 13.8. Импортирование составляющих обычного (не синглтон) объекта

```
def showFruit(fruit: Fruit) = {
  import fruit._
  println(name + "s are " + color)
}
```

Метод `showFruit` импортирует все составляющие его параметра `fruit`, относящегося к типу `Fruit`. Следующая инструкция `println` может непосредственно ссылаться на `name` и `color`. Эти две ссылки являются эквивалентами ссылок `fruit.name` и `fruit.color`. Такой синтаксис пригодится, в частности, при использовании объектов в качестве модулей. Соответствующее описание будет дано в главе 29.

Гибкость импортирования в Scala

Директива import работает в Scala более гибко, чем в Java. Эта гибкость характеризуется тремя принципиальными отличиями. Импорт кода в Scala:

может появляться где угодно;

позволяет кроме пакетов ссылаться на объекты (синглтон или

обычные);

позволяет изменять имена или скрывать некоторые из импортированных составляющих.

Еще один из факторов гибкости импорта кода в Scala заключается в возможности импортировать пакеты как таковые, без их непакетированного наполнения. Смысл в этом будет только в том случае, если предполагается, что в пакете заключены другие пакеты. Например, в листинге 13.9 импортируется пакет `java.util.regex`. Благодаря этому `regex` можно использовать с указанием его простого имени. Чтобы обратиться к синглтон-объекту `Patterns` из пакета `java.util.regex`, можно, как показано в листинге 13.9, просто воспользоваться идентификатором `regex.Pattern`.

Листинг 13.9. Импортирование имени пакета

```
import java.util.regex

class AStarB {
  // Обращение к java.util.regex.Pattern
  val pat = regex.Pattern.compile("a*b")
}
```

При импортировании кода Scala позволяет также переименовывать или скрывать элементы. Это делается путем заключения импорта в фигурные скобки с указанием перед получившимся в результате этого блоком того объекта, из которого импортируются его элементы. Рассмотрим несколько примеров:

- `import Fruits.{Apple, Orange}`

Здесь из объекта `Fruits` импортируются только его составляющие `Apple` и `Orange`.

- `import Fruits.{Apple => McIntosh, Orange}`

Здесь из объекта `Fruits` импортируются две составляющие, `Apple` и `Orange`. Но объект `Apple` переименовывается в `McIntosh`, поэтому к нему можно обращаться либо как `Fruits.Apple`, либо как `McIntosh`. Директива переименования всегда имеет вид <исходное_имя> => <новое_имя>.

- `import java.sql.{Date => SDate}`

Здесь под именем `SDate` импортируется класс данных SQL, чтобы можно было в то же время импортировать обычный класс данных Java просто как `Date`.

- `import java.{sql => S}`

Здесь под именем `S` импортируется пакет `java.sql`, чтобы можно было воспользоваться кодом вида `S.Date`.

- `import Fruits.{_}`

Здесь импортируются все составляющие объекта `Fruits`. Это означает то же самое, что и `import Fruits._`.

- `import Fruits.{Apple => McIntosh, _}`

Здесь импортируются все составляющие объекта `Fruits`, но `Apple` переименовывается в `McIntosh`.

- `import Fruits.{Pear => _, _}`

  Здесь импортируются все составляющие объекта `Fruits`, за исключением `Pear`. Директива вида <исходное_имя> => _ исключает <исходное_имя> из импортируемых имен. В определенном смысле переименование чего-либо в _ означает полное сокрытие переименованной составляющей. Это помогает избегать неоднозначностей. Предположим, имеются два пакета, `Fruits` и `Notebooks`, и в каждом из них определен класс `Apple`. Если нужно получить только ноутбук под названием Apple, а не фрукт, можно воспользоваться двумя импортами по запросу:

  ```
  import Notebooks._
  import Fruits.{Apple => _, _}
  ```

  Будут импортированы все составляющие `Notebooks` и все составляющие `Fruits`, за исключением `Apple`.

Эти примеры демонстрируют поразительную гибкость, предлагаемую Scala в вопросах избирательного импортирования составляющих, возможно, даже под другими именами. Таким образом, директива `import` может состоять из следующих селекторов:

- простого имени x, которое включается в набор импортируемых имен;
- директивы переименования x => y. Составляющая по имени x будет видна под именем y;
- директивы сокрытия x => _. Имя x исключается из набора импортируемых имен;
- элемента «забрать все» (catch-all) _. Импортируются все

элементы, за исключением тех составляющих, которые были упомянуты в предыдущей директиве. Если указан элемент «забрать все», то в списке селекторов импортирования он должен стоять последним.

Самые простые `import`-директивы, показанные в начале данного раздела, могут рассматриваться как специальные сокращения директив `import` с селекторами. Например, `import p._` является эквивалентом `import p.{_}`, а `import p.n` — эквивалентом `import p.{n}`.

13.4. Подразумеваемое импортирование

Подразумевается, что Scala добавляет импортируемый код в каждую программу. По сути, происходит то, что было бы при добавлении в самое начало каждого исходного файла с расширением **.scala** следующих трех директив `import`:

```
import java.lang._  // забрать все из пакета java.lang
import scala._      // забрать все из пакета scala
import Predef._     // забрать все из пакета Predef
```

В пакете `java.lang` содержатся стандартные классы Java. Он всегда подразумеваемо импортируется в исходные файлы Scala[88]. Безусловное импортирование `java.lang` позволяет вам, к примеру, использовать вместо `java.lang.Thread` просто идентификатор `Thread`.

Теперь уже вряд ли приходится сомневаться в том, что в пакете `scala` находится стандартная библиотека Scala, в которой содержатся многие самые востребованные классы и объекты. Благодаря подразумеваемости импортирования пакета `scala` можно, к примеру, вместо `scala.List` указать просто `List`.

В объекте `Predef` содержится множество определений типов, методов и подразумеваемых преобразований, которые обычно используются в программах на Scala. Благодаря подразумеваемости импортирования `Predef` можно, к примеру, вместо `Predef.assert` воспользоваться просто идентификатором `assert`.

Эти три директивы `import` трактуются особым образом, позволяющим тому импорту, который указан позже, затенять ранее указанный импорт. К примеру, класс `StringBuilder` определен как в пакете `scala`, так и, начиная с версии Java 1.5, в пакете `java.lang`. Поскольку импорт `scala` затеняет импорт `java.lang`, простое имя `StringBuilder` будет ссылаться на `scala.StringBuilder`, а не на `java.lang.StringBuilder`.

13.5. Модификаторы доступа

Элементы пакетов, классов или объектов могут быть помечены модификаторами доступа `private` и `protected`. Они ограничивают доступ к элементам, позволяя обращаться к ним только из определенных областей кода. Трактовка модификаторов доступа Scala примерно соответствует той, что принята в Java, но при этом имеется ряд весьма важных отличий, рассматриваемых в данном разделе.

Закрытые элементы

Закрытые элементы трактуются в Scala точно так же, как и в Java. Элемент с пометкой `private` виден только внутри класса или объекта, в котором содержится определение этого элемента. В Scala это правило распространяется и на внутренние классы. Данная трактовка более последовательна, но отличается от той, что принята в Java. Рассмотрим пример, показанный в листинге 13.10.

Листинг 13.10. Отличие закрытого доступа в Scala от такого же доступа в Java

```
class Outer {
  class Inner {
    private def f() = { println("f") }
    class InnerMost {
      f() // Вполне допустимо
    }
  }
    (new Inner).f() // ошибка: доступ к f
отсутствует
}
```

В Scala обращение `(new Inner).f()` недопустимо, поскольку закрытое объявление `f` сделано в `Inner`, а попытка обращения делается не из класса `Inner`. В отличие от этого, первое обращение к `f` в классе `InnerMost` вполне допустимо, поскольку оно содержится в теле класса `Inner`. В Java допустимы оба обращения, поскольку в этом языке разрешается обращение из внешнего класса к закрытым составляющим его внутренних классов.

Защищенные элементы

Доступ к защищенным элементам в Scala также менее свободен, чем в Java. В Scala обратиться к защищенному элементу можно только из подклассов того класса, в котором этот элемент был определен. В Java обращение возможно и из других классов того же самого пакета. В Scala есть еще один способ достижения того же самого эффекта[89], поэтому модификатор `protected` можно оставить без изменений. Защищенные виды доступа показаны в листинге 13.11.

Листинг 13.11. Отличие защищенного доступа в Scala от такого же доступа в Java

```
package p {
  class Super {
    protected def f() = { println("f") }
  }
  class Sub extends Super {
    f()
  }
  class Other {
       (new Super).f() // ошибка: доступ к f
отсутствует
  }
}
```

В листинге 13.11 обращение к `f` в классе `Sub` вполне допустимо, поскольку объявление `f` было сделано с модификатором `protected` в `Super`, а `Sub` является подклассом `Super`. В отличие от этого, обращение к `f` в `Other` недопустимо, поскольку `Other` не является наследником `Super`. В Java последнее обращение все равно будет разрешено, поскольку `Other` находится в том же самом пакете, что и `Sub`.

Открытые элементы

В Scala нет явного модификатора для открытых элементов: любой элмент, не помеченный как `private` или `protected`, является открытым. К открытым элементам можно обращаться откуда угодно.

Область защиты

Модификаторы доступа в Scala могут дополняться

спецификаторами. Модификатор вида `private[X]` или `protected[X]` означает, что доступ закрыт или защищен вплоть до *X*, где *X* определяет некий внешний пакет, класс или синглтон-объект.

Специфицированные модификаторы доступа позволяют весьма четко обозначить границы управления видимостью. В частности, они позволяют выразить понятия доступности, имеющиеся в Java, такие как закрытость пакета, защищенность пакета или закрытость вплоть до самого внешнего класса, которые невозможно выразить напрямую с помощью простых модификаторов, используемых в Scala. Но, кроме этого, они позволяют выразить правила доступности, которые не могут быть выражены в Java.

В листинге 13.12 представлен пример с использованием множества спецификаторов доступа. Здесь класс `Navigator` помечен как `private[bobsrockets]`. Это означает, что он имеет область видимости, охватывающую все классы и объекты, которые содержатся в пакете `bobsrockets`. В частности, доступ к `Navigator` разрешен в объекте `Vehicle`, поскольку `Vehicle` содержится в пакете `launch`, который в свою очередь содержится в пакете `bobsrockets`. В то же время весь код, который находится за пределами пакета `bobsrockets`, не может получить доступ к классу `Navigator`.

Листинг 13.12. Придание гибкости областям защиты с помощью спецификаторов доступа

```
package bobsrockets

package navigation {
  private[bobsrockets] class Navigator {
    protected[navigation] def useStarChart() = {}
    class LegOfJourney {
```

```
      private[Navigator] val distance = 100
    }
    private[this] var speed = 200
  }
}
package launch {
  import navigation._
  object Vehicle {
    private[launch] val guide = new Navigator
  }
}
```

Этот прием особенно полезен при разработке крупных проектов, содержащих несколько пакетов. Он позволяет определять элементы, видимость которых распространяется на несколько подчиненных пакетов проекта, оставляя их невидимыми для клиентов, являющихся внешними по отношению к данному проекту. Применить подобный прием в Java не представляется возможным. Там, как только определение переходит границу своего пакета, элемент становится видимым повсеместно.

Разумеется, действие спецификатора закрытости может распространяться и на непосредственно окружающий пакет. В листинге 13.12 показан пример модификатора доступа `guide` в объекте `Vehicle`. Такой модификатор доступа является эквивалентом имеющегося в Java доступа, ограниченного пределами одного пакета.

Все спецификаторы также могут применяться к модификатору `protected` со значениями, аналогичными тем, с которыми они применяются к модификатору `private`. То есть модификатор `protected[`*X*`]` в классе `C` позволяет получить доступ к определению с такой пометкой во всех подклассах `C`, а также во внешнем пакете, классе или объекте с названием *X*. Например,

метод `useStarChart` в листинге 13.12 доступен из всех подклассов `Navigator`, а также из всего кода, содержащегося во внешнем пакете `navigation`. В результате получается точное соответствие значению модификатора `protected` в Java.

Спецификаторы модификатора `private` могут также ссылаться на окружающий (внешний) класс или объект. Например, показанная в листинге 13.12 переменная `distance` в классе `LegOfJourney` имеет пометку `private[Navigator]`, следовательно, она видима из любого места в классе `Navigator`. Тем самым ей придаются такие же возможности видимости, как и закрытым составляющим внутренних классов в Java. Модификатор `private[`*C*`]`, где *C* является самым внешним классом, аналогичен простому модификатору `private` в Java.

И наконец, в Scala имеется также модификатор доступа, обладающий еще большей степенью ограниченности, чем `private`. Определение с пометкой `private[this]` доступно только внутри того же самого объекта, в котором содержится это определение. Такое определение называется закрытым внутри объекта. Например, в листинге 13.12 закрытым внутри объекта является определение `speed` в классе `Navigator`. Это означает, что любое обращение должно быть сделано не только внутри класса `Navigator`, но и внутри именно этого экземпляра класса `Navigator`. Таким образом, внутри класса `Navigator` будут вполне допустимы обращения `speed` и `this.speed`.

Но следующее обращение не будет позволено, даже если оно появится внутри класса `Navigator`:

```
val other = new Navigator
other.speed // эта строка кода не пройдет компиляцию
```

Пометка элемента модификатором `private[this]` служит гарантией того, что он не будет виден из других объектов того же самого класса. Это может пригодиться для документирования.

Иногда такой подход также позволяет создавать более универсальные варианты аннотаций (подробности рассматриваются в разделе 19.7).

В качестве резюме в табл. 13.1 приведен список действий спецификаторов модификатора `private`. В каждой строке показан модификатор `private` со спецификатором и раскрыто его значение при применении в отношении переменной `distance`, объявленной в классе `LegOfJourney` в листинге 13.12.

Таблица 13.1. Действия спецификаторов private в отношении LegOfJourney.distance

Спецификатор	Действие
Без указания модификатора доступа	Открытый доступ
private[bobsrockets]	Доступ в пределах внешнего пакета
private[navigation]	Аналог имеющейся в Java видимости в пределах пакета
private[Navigator]	Аналог имеющегося в Java модификатора private
private[LegOfJourney]	Аналог имеющегося в Scala модификатора private
private[this]	Доступ только из того же самого объекта

Видимость и объекты-спутники

В Java статические элементы и элементы экземпляра принадлежат одному и тому же классу, поэтому модификаторы доступа применяются к ним одинаково. Вы уже видели, что в Scala статических элементов нет, вместо них может быть объект-спутник, содержащий элементы, существующие в единственном экземпляре. Например, в листинге 13.13 объект `Rocket` является спутником класса `Rocket`.

Листинг 13.13. Обращение к закрытым элементам классов-спутников и объектов-спутников

```
class Rocket {
  import Rocket.fuel
  private def canGoHomeAgain = fuel > 20
}

object Rocket {
  private def fuel = 10
  def chooseStrategy(rocket: Rocket) = {
    if (rocket.canGoHomeAgain)
      goHome()
    else
      pickAStar()
  }
  def goHome() = {}
  def pickAStar() = {}
}
```

Что касается закрытого или защищенного доступа, то в правилах доступа, действующих в Scala, объектам-спутникам и классам-спутникам даются особые привилегии. Класс делится всеми своими правами доступа со своим объектом-спутником и наоборот. В частности, объект может обращаться ко всем закрытым элементам своего класса-спутника точно так же, как класс может обращаться ко всем закрытым элементам своего объекта-спутника.

Например, в листинге 13.13 класс `Rocket` может обращаться к методу `fuel`, который объявлен закрытым в объекте `Rocket`. Аналогично этому объект `Rocket` может обращаться к закрытому методу `canGoHomeAgain` в классе `Rocket`.

Одно из исключений, нарушающих аналогию между Scala и Java, касается защищенных статических элементов. Защищенный статический элемент Java-класса `C` может быть доступен во всех подклассах `C`. В отличие от этого, в наличии защищенного

элемента в объекте-спутнике нет никакого смысла, поскольку у синглтон-объектов нет никаких подклассов.

13.6. Объекты пакетов

До сих пор единственным встречающимся вам кодом, добавляемым к пакетам, были классы, трейты и отдельные объекты. Они, несомненно, являются наиболее распространенными определениями, помещаемыми на самом верхнем уровне пакета. Но Scala не ограничивает вас только этим перечнем — любые виды определений, которые можно помещать внутри класса, могут присутствовать и на верхнем уровне пакета. На самый верхний уровень пакета можно смело помещать любой вспомогательный метод, который хотелось бы иметь в области видимости всего пакета.

Для этого поместите определение в *объект пакета*. В каждом пакете разрешается иметь один объект пакета. Любые определения, помещенные в объекте пакета, считаются элементами самого пакета.

Пример показан в листинге 13.14. В файле **package.scala** содержится объект пакета для пакета `bobsdelights`. Синтаксически объект пакета очень похож на одно из ранее показанных в этой главе пакетирований в фигурных скобках. Единственное отличие заключается в том, что в него включено ключевое слово `object`. Это *объект* пакета, а не сам *пакет*. Содержимое, заключенное в фигурные скобки, может включать любые нужные вам определения. В данном случае объект пакета включает вспомогательный метод `showFruit` из листинга 13.8.

Листинг 13.14. Объект пакета

```
// В файле bobsdelights/package.scala
package object bobsdelights {
```

```
  def showFruit(fruit: Fruit) = {
    import fruit._
    println(name + "s are " + color)
  }
}

// В файле PrintMenu.scala
package printmenu
import bobsdelights.Fruits
import bobsdelights.showFruit

object PrintMenu {
  def main(args: Array[String]) = {
    for (fruit <- Fruits.menu) {
      showFruit(fruit)
    }
  }
}
```

При наличии этого определения любой другой код в любом пакете может импортировать метод точно так же, как он мог бы импортировать класс. Например, в листинге 13.14 показан автономный объект `PrintMenu`, который находится в другом пакете. `PrintMenu` может импортировать вспомогательный метод `showFruit` точно так же, как он импортировал бы класс `Fruit`.

Забегая вперед, следует отметить, что имеются и другие примеры использования объектов пакетов для тех разновидностей определений, которые вам еще не были показаны. Объекты пакетов часто используются для содержания псевдонимов типов, предназначенных для всего пакета (глава 20) и подразумеваемых преобразований (глава 21). В пакете верхнего уровня `scala` имеется объект пакета, чьи определения доступны всему коду Scala. Объекты пакетов компилируются в файлы классов с именами

package.class, размещаемыми в каталоге **package**, который они дополняют. Им будет полезно придерживаться того же соглашения, которое действует в отношении исходных файлов. Таким образом, исходный код объекта пакета `bobsdelights` из листинга 13.14 было бы разумно поместить в файл **package.scala**, размещаемый в каталоге **bobsdelights**.

Резюме

В главе были показаны основные конструкции, предназначенные для разбиения программы на пакеты. Это дает вам простую и полезную разновидность модульности и позволяет работать с весьма большими объемами кода, не допуская взаимного влияния различных его частей. Существующая в Scala система по духу аналогична пакетированию, используемому в Java, но с некоторыми отличиями: в Scala проявляется более последовательный или же более универсальный подход.

Забегая вперед: в главе 29 дается описание более гибкой системы модулей, разбиваемой на пакеты. Данный подход, кроме того что позволяет делить код на несколько пространств имен, дает модулям возможность взаимного наследования и параметризации. В следующей главе наше внимание будет переключено на утверждения и блочное тестирование.

87 Этот стиль с использованием нескольких директив package без фигурных скобок называется объявлением цепочки пакетов.

88 Изначально имелась также реализация Scala на платформе .NET, где вместо этого импортировалось пространство имен System, .NET-аналог пакета java.lang.

89 Используя спецификаторы, рассматриваемые далее в разделе «Область защиты».

14. Утверждения и тесты

Утверждения и тесты являются двумя важными способами проверки правильности поведения разработанных вами программных средств. В данной главе будут показаны несколько вариантов для их создания и запуска, доступных в Scala.

14.1. Утверждения

Утверждения в Scala создаются в виде вызовов предопределенного метода `assert`[90]. Выражение `assert(условие)` выдает ошибку `AssertionError`, если условие не соблюдается. Существует также версия `assert`, использующая два аргумента: выражение `assert(условие, объяснение)` тестирует условие и, если оно не соблюдается, выдает ошибку `AssertionError`, в сообщении о которой содержится заданное объяснение. Типом объяснения является `Any`, поэтому в качестве объяснения может передаваться любой объект. Чтобы поместить в `AssertionError` строковое объяснение, метод `assert` будет вызывать в отношении этого объекта метод `toString`. Например, в метод по имени `above`, относящийся к классу `Element` и показанный в листинге 10.13, `assert` можно поместить после вызовов `widen`, чтобы убедиться, что расширенные элементы имеют одинаковую ширину. Этот вариант представлен в листинге 14.1.

Листинг 14.1. Использование утверждения

```
def above(that: Element): Element = {
  val this1 = this widen that.width
  val that1 = that widen this.width
  assert(this1.width == that1.width)
  elem(this1.contents ++ that1.contents)
```

```
}
```

Для выполнения той же задачи можно выбрать и другой способ и проверить ширину в конце метода `widen`, непосредственно перед возвращением значения. Этого можно добиться, сохранив результат в `val`-переменной, выполняя утверждение применительно к результату с последующим указанием `val`-переменной, чтобы результат возвращался в том случае, если утверждение было успешно подтверждено. Но, как показано в листинге 14.2, это можно сделать более выразительно, воспользовавшись довольно удобным методом `ensuring` из синглтон-объекта `Predef`.

Листинг 14.2. Использование метода ensuring для утверждения результата выполнения функции

```
private def widen(w: Int): Element =
  if (w <= width)
    this
  else {
    val left = elem(' ', (w - width) / 2, height)
     var right = elem(' ', w - width - left.width,
height)
    left beside this beside right
} ensuring (w <= _.width)
```

Благодаря подразумеваемому преобразованию метод `ensuring` может использоваться с результатом любого типа. Хотя в данном коде все выглядит так, будто `ensuring` вызван в отношении результата выполнения метода `widen`, имеющего тип `Element`, на самом деле `ensuring` вызван в отношении типа, в который `Element` был автоматически преобразован. Метод `ensuring` получает один аргумент, функцию-предикат, которая берет тип

результата, возвращает булево значение и передает результат предикату. Если предикат возвращает `true`, метод `ensuring` возвращает результат, в противном случае метод выдаст ошибку `AssertionError`.

В данном примере предикат имеет вид `w <= _.width`. Знак подчеркивания является заместителем для одного аргумента, передаваемого предикату, а именно результата типа `Element`, получаемого от метода `widen`. Если ширина, переданная в виде `w` методу `widen`, меньше ширины результата типа `Element` или равна ей, предикат будет вычислен в `true` и результатом `ensuring` будет объект типа `Element`, в отношении которого он был вызван. Поскольку это выражение в методе `widen` последнее, его результат типа `Element` и будет значением, возвращаемым самим методом `widen`.

Утверждения могут включаться и выключаться с помощью ключей командной строки JVM `-ea` и `-da`. При включении каждое утверждение служит небольшим тестом, использующим фактические данные, вычисленные при выполнении программы. Далее в главе основное внимание будет уделено созданию внутренних тестов, предоставляющих свои собственные тестовые данные и выполняемых независимо от приложения.

14.2. Тестирование в Scala

В Scala имеется множество вариантов тестирования, начиная с установленного инструментария Java, такого как JUnit и TestNG, и заканчивая инструментарием, написанным на Scala, например ScalaTest, specs2 и ScalaCheck. В оставшейся части главы будет дан краткий обзор этого инструментария, который начнется с рассмотрения ScalaTest.

ScalaTest является наиболее гибкой средой тестирования в Scala, это средство можно легко настроить на решение различных проблем. Гибкость ScalaTest означает, что команды могут

использовать тот стиль тестирования, который более полно отвечает их потребностям. Например, для команд, знакомых с JUnit, наиболее знакомым и удобным станет стиль FunSuite. Пример показан в листинге 14.3.

Листинг 14.3. Написание тестов с использованием FunSuite

```
import org.scalatest.FunSuite
import Element.elem

class ElementSuite extends FunSuite {

  test("elem result should have passed width") {
    val ele = elem('x', 2, 3)
    assert(ele.width == 2)
  }
}
```

Центральной концепцией в ScalaTest является *набор*, то есть коллекция тестов. Тестом может выступать любой код с именем, который может быть запущен с получением удачного результата или сбоя, отложен или отменен. Центральным блоком композиции в ScalaTest является трейт Suite. В нем объявляются методы жизненного цикла, определяющие исходный способ запуска тестов, который может быть переопределен под нужные способы написания и запуска тестов.

В ScalaTest предлагаются *стилевые трейты*, расширяющие Suite и переопределяющие методы жизненного цикла для поддержания различных стилей тестирования. В этой среде также предоставляются *подмешиваемые трейты*, переопределяющие методы жизненного цикла таким образом, чтобы они отвечали конкретным потребностям тестирования. Классы тестов определяются сочетанием стиля Suite и подмешиваемых трейтов,

а тестовые наборы — путем составления экземпляров Suite.

Примером стиля тестирования является FunSuite, расширенный тестовым классом, показанным в листинге 14.3. Слово «Fun» в FunSuite означает функцию, а test является определенным в FunSuite методом, который вызывается первичным конструктором ElementSuite. Вы указываете название теста в виде строки в круглых скобках и сам код теста, помещаемый в фигурные скобки. Код теста является функцией, передаваемой методу test в виде параметра до востребования, которая регистрирует его для последующего выполнения.

Среда ScalaTest интегрирована в общие инструментальные средства сборки, такие как sbt и Maven, и в IDE-среды, такие как IntelliJIDEA и Eclipse. Suite можно также запустить непосредственно через приложение ScalaTest Runner или из интерпретатора Scala, просто вызвав в отношении него метод execute. Соответствующий пример выглядит следующим образом:

```
scala> (new ElementSuite).execute()
ElementSuite:
- elem result should have passed width
```

Все стили ScalaTest, включая FunSuite, предназначены для стимуляции написания специализированных тестов с осмысленными названиями. Кроме того, все стили создают вывод, похожий на спецификацию, которая может облегчить общение между заинтересованными сторонами. Выбранным вами стилем предписывается только то, как будут выглядеть объявления ваших тестов. Все остальное в ScalaTest работает одинаково, независимо от выбранного стиля[91].

14.3. Информативные отчеты об ошибках

В тесте, показанном в листинге 14.3, предпринимается попытка

создания элемента с шириной, равной 2, и высказывается утверждение, что ширина получившегося элемента действительно равна двум. Если утверждение не подтвердится, отчет об ошибке будет включать имя файла и номер строки с неоправдавшимся утверждением, а также информативное сообщение об ошибке:

```
scala> val width = 3
width: Int = 3

scala> assert(width == 2)
org.scalatest.exceptions.TestFailedException:
    3 did not equal 2
```

Чтобы обеспечить выведение содержательных сообщений об ошибках при неверных утверждениях, ScalaTest в ходе компиляции анализирует выражения, переданные каждому вызову утверждения. Если предпочтительнее увидеть более подробную информацию о неоправдавшихся утверждениях, можно воспользоваться имеющимся в ScalaTest средством DiagrammedAssertions, чьи сообщения об ошибках показывают схему выражения, переданного утверждению `assert`:

```
scala> assert(List(1, 2, 3).contains(4))
org.scalatest.exceptions.TestFailedException:

  assert(List(1, 2, 3).contains(4))
         |    |  |  |  |        |
         |    1  2  3  false    4
         List(1, 2, 3)
```

Имеющиеся в ScalaTest методы `assert` не проводят в сообщениях об ошибках различий между фактическим и ожидаемым результатами. Они просто показывают, что левый операнд не эквивалентен правому, или показывают значения на

схеме. Если нужно подчеркнуть различия между фактическим и ожидаемым результатами, можно воспользоваться имеющимся в ScalaTest альтернативным методом `assertResult`:

```
assertResult(2) {
  ele.width
}
```

С помощью данного выражения показывается, что от кода в фигурных скобках ожидается результат 2. Если после выполнения этого кода получится 3, то в отчете об ошибке тестирования будет показано сообщение `Expected 2, but got 3` («Ожидалось 2, но получено 3»).

Если нужно проверить, что метод выдает ожидаемое исключение, можно воспользоваться имеющимся в ScalaTest методом `assertThrows`:

```
assertThrows[IllegalArgumentException] {
  elem('x', -2, 3)
}
```

Если код в фигурных скобках выдает не то исключение, которое ожидалось, или вообще не выдает его, метод `assertThrows` тут же завершит свою работу с выдачей исключения `TestFailedException`. А в отчете об ошибке будет сообщение с полезной для вас информацией:

```
Expected IllegalArgumentException to be thrown,
  but NegativeArraySizeException was thrown.
```

Но если выполнение кода внезапно завершится с экземпляром переданного класса исключения, управление из метода `assertThrows` будет возвращено в обычном порядке. Если потребуется дальнейшее исследование ожидаемого исключения, можно вместо `assertThrows` воспользоваться методом перехвата

intercept. Метод intercept работает аналогично методу asassertThrows, за исключением того, что при выдаче ожидаемого исключения intercept возвращает это исключение:

```
val caught =
  intercept[ArithmeticException] {
    1 / 0
  }

assert(caught.getMessage == "/ by zero")
```

Короче говоря, имеющиеся в ScalaTest утверждения стараются выдать полезные сообщения об ошибках, способные помочь вам диагностировать и устранить проблемы в вашем коде.

14.4. Использование тестов в качестве спецификаций

В стиле тестирования при *разработке через реализацию поведения* (behavior-driven development (BDD)) основной упор делается на написание легко воспринимаемых человеком спецификаций расширенного поведения кода и сопутствующих тестов, проверяющих наличие у кода такого поведения. В ScalaTest включены несколько трейтов, содействующих этому стилю тестирования. Пример использования одного такого трейта, FlatSpec, показан в листинге 14.4.

Листинг 14.4. Спецификация и тестирование поведения с помощью ScalaTest FlatSpec

```
import org.scalatest.FlatSpec
import org.scalatest.Matchers
import Element.elem
```

```
class ElementSpec extends FlatSpec with Matchers {
  "A UniformElement" should
      "have a width equal to the passed value" in
{
    val ele = elem('x', 2, 3)
    ele.width should be (2)
  }
   it should "have a height equal to the passed
value" in {
    val ele = elem('x', 2, 3)
    ele.height should be (3)
  }

   it should "throw an IAE if passed a negative
width" in {
        an [IllegalArgumentException] should be
thrownBy {
      elem('x', -2, 3)
    }
  }
}
```

При использовании `FlatSpec` тесты создаются в виде *директив спецификации*. Сначала в виде строки пишется название тестируемого субъекта (`"A UniformElement"` в листинге 14.4), затем `should`, или `must`, или `can` (что означает «обязан», или «должен», или «может» соответственно), потом строка, обозначающая характер поведения, требуемого от субъекта, а после — ключевое слово `in`. В фигурных скобках, стоящих после `in`, пишется код, тестирующий указанное поведение. В последующих директивах, чтобы сослаться на самый последний субъект, можно написать `it`. При выполнении `FlatSpec` этот трейт будет запускать каждую директиву спецификатора в виде теста ScalaTest.

FlatSpec (и другие спецификационные трейты, имеющиеся в ScalaTest) генерирует вывод, который при запуске читается как спецификация. Например, вот на что будет похож вывод, если запустить ElementSpec из листинга 14.4 в интерпретаторе:

```
scala> (new ElementSpec).execute()
A UniformElement
- should have a width equal to the passed value
- should have a height equal to the passed value
- should throw an IAE if passed a negative width
```

В листинге 14.4 также показан имеющийся в ScalaTest предметно-ориентированный язык (domain-specific language (DSL)) *выявления соответствий*. Подмешиванием трейта Matchers можно создавать утверждения, которые при чтении больше похожи на естественный язык. В имеющемся в ScalaTest DSL-языке предоставляется множество средств выявления соответствий, а кроме этого, он позволяет определять новые предоставленные пользователем средства выявления соответствий, содержащие сообщения об ошибках. Средства выявления соответствий,- показанные в листинге 14.4, включают синтаксис should be и an [...] should be thrownBy { ...}. Как вариант, если предпочтение отдается глаголу must, а не глаголу should, можно подмешать MustMatchers. Например, подмешивание MustMatchers позволит вам создавать следующие выражения:

```
result must be >= 0
map must contain key 'c'
```

Если последнее утверждение не подтвердится, будет показано сообщение об ошибке следующего вида:

```
Map('a' -> 1, 'b' -> 2) did not contain key 'c'
```

Среда тестирования specs2 — средство с открытым кодом,

написанное на Scala Эриком Торребо (Eric Torreborre), — также поддерживает BDD-стиль тестирования, но с другим синтаксисом. Например, specs2 можно использовать для создания теста, показанного в листинге 14.5.

Листинг 14.5. Спецификация и тестирование поведения с использованием среды specs2

```
import org.specs2._
import Element.elem

object ElementSpecification extends Specification
{
  "A UniformElement" should {
    "have a width equal to the passed value" in {
      val ele = elem('x', 2, 3)
      ele.width must be_==(2)
    }
    "have a height equal to the passed value" in {
      val ele = elem('x', 2, 3)
      ele.height must be_==(3)
    }
    "throw an IAE if passed a negative width" in {
      elem('x', -2, 3) must
        throwA[IllegalArgumentException]
    }
  }
}
```

В specs2, как и в ScalaTest, существует DSL-язык выявления соответствий. Некоторые примеры работы средств выявления соответствий, имеющихся в specs2, показаны в листинге 14.5 в

строках, содержащих `must be_==` и `must throwA`[92]. Среду specs2 можно использовать в автономном режиме, но она также интегрируется со ScalaTest и JUnit, поэтому specs2-тесты можно запускать и с этими инструментальными средствами.

Одной из основных идей BDD является то, что тесты могут использоваться для содействия обмену мнениями между людьми, принимающими решения о характере поведения программных средств, людьми, разрабатывающими программные средства, и людьми, определяющими степень завершенности и работоспособности программных средств. Хотя в этом ключе могут использоваться любые стили, имеющиеся в ScalaTest или specs2, в ScalaTest есть специально разработанный для этого стиль `FeatureSpec`. Пример его использования показан в листинге 14.6.

Листинг 14.6. Использование тестов для содействия обмену мнениями среди всех заинтересованных сторон

```
import org.scalatest._

class TVSetSpec extends FeatureSpec with
GivenWhenThen {

  feature("TV power button") {
    scenario("User presses power button when TV is
off") {
      Given("a TV set that is switched off")
      When("the power button is pressed")
      Then("the TV should switch on")
      pending
    }
  }
}
```

FeatureSpec разработан с целью направления в нужное русло обсуждений предназначения программных средств: вам следует выявить специфические *особенности*, а затем дать им точное определение в понятиях *сценариев*. Сосредоточиться на переговорах об особенностях отдельно взятых сценариев помогают методы Given, When и Then, предоставляемые трейтом GivenWhenThen. Вызов pending в самом конце показывает, что как тест, так и само поведение не реализованы, имеется лишь спецификация. Как только будут реализованы все тесты и конкретные действия, тесты будут пройдены и требования можно будет посчитать выполненными.

14.5. Тестирование на основе свойств

Еще одним полезным средством тестирования для Scala является ScalaCheck — среда с открытым кодом, созданная Рикардом Нильссоном (Rickard Nilsson). ScalaCheck позволяет указывать свойства, которыми должен обладать тестируемый код. Для каждого свойства ScalaCheck создает данные и выдает утверждения, проверяющие наличие тех или иных свойств. Пример использования ScalaCheck из ScalaTest WordSpec, в который подмешан трейт PropertyChecks, показан в листинге 14.7.

Листинг 14.7. Написание тестов на основе проверки наличия свойств с помощью ScalaCheck

```
import org.scalatest.WordSpec
import org.scalatest.prop.PropertyChecks
import org.scalatest.MustMatchers._
import Element.elem

class ElementSpec extends WordSpec with
PropertyChecks {
```

```
    "elem result" must {
      "have passed width" in {
        forAll { (w: Int) =>
          whenever (w > 0) {
            elem('x', w, 3).width must equal (w)
          }
        }
      }
    }
  }
```

WordSpec является классом стиля, имеющимся в ScalaTest. Трейт PropertyChecks предоставляет несколько методов forAll, позволяющих смешивать тесты на основе проверки наличия свойств с традиционными тестами на основе утверждений или на основе выявления соответствий. В данном примере проверяется наличие свойства, которым должен обладать фабричный метод elem. Свойства ScalaCheck выражены в виде значений функций, получающих в качестве параметров данные, необходимые для утверждений о наличии свойств. Эти данные будут генерироваться ScalaCheck. В свойстве, показанном в листинге 14.7, данными является целое число, названное w, которое представляет ширину. Внутри тела функции представлен следующий код:

```
whenever (w > 0) {
  elem('x', w, 3).width must equal (w)
}
```

Директива whenever указывает на то, что при каждом вычислении левостороннего выражения в true правостороннее выражение также должно содержать true. Таким образом, в данном случае выражение в блоке должно содержать true всякий раз, когда w больше нуля. А правостороннее выражение будет выдавать true, если ширина, переданная фабричному методу

elem, будет равна ширине Element-объекта, возвращенного фабричным методом.

При таком небольшом объеме кода ScalaCheck сгенерирует, возможно, сотни пробных значений, проверяя каждое из них и выискивая значение, для которого свойство не соблюдается. Если свойство соблюдается для каждого значения, испытанного с помощью ScalaCheck, тест будет пройден. В противном случае он внезапно завершится с выдачей исключения TestFailedException, которое содержит информацию, включающую значение, вызвавшее сбой.

14.6. Подготовка и проведение тестов

Во всех средах, упомянутых в данной главе, имеется некий механизм для подготовки и проведения тестов. В этом разделе будет дан краткий обзор того подхода, который используется в ScalaTest. Чтобы получить подробное описание любой из этих сред, нужно обратиться к их документации.

Подготовка больших наборов тестов в ScalaTest проводится путем вложения Suite-наборов в Suite-наборы. При выполнении Suite-набор запустит не только свои тесты, но и все тесты вложенных в него Suite-наборов. Вложенные Suite-наборы в свою очередь выполнят тесты вложенных в них Suite-наборов и т. д. Таким образом, большой набор тестов будет представлен деревом Suite-объектов. При выполнении в этом дереве корневого Suite-объекта выполнятся все имеющиеся в нем Suite-объекты.

Наборы можно вкладывать самостоятельно или автоматически. Чтобы вложить их вручную, вам нужно либо переопределить метод nestedSuites в своих Suite-классах, либо передать предназначенные для вложения Suite-объекты в конструктор класса Suites, который для этих целей предоставляет ScalaTest. Для автоматического вложения имена пакетов передаются

имеющемуся в ScalaTest средству Runner, которое определит `Suite`-объекты автоматически, вложит их ниже корневого `Suite` и выполнит корневой `Suite`.

Имеющееся в ScalaTest приложение Runner можно вызвать из командной строки или через такие средства сборки, как sbt, maven или ant. Проще всего вызвать Runner в командной строке через приложение theorg.scalatest.run. Оно ожидает полностью указанного тестового класса. Например, для запуска тестового класса, показанного в листинге 14.6, его нужно скомпилировать с помощью следующей команды:

```
$ scalac -cp scalatest.jar TVSetSpec.scala
```

Затем его можно будет запустить, воспользовавшись командой

```
$ scala -cp scalatest.jar org.scalatest.run
TVSetSpec
```

Используя ключ `-cp`, вы помещаете имеющийся в ScalaTest JAR-файл в путь класса. (При скачивании в имя JAR-файла будут включены встроенные номера версий Scala и ScalaTest.) Следующий элемент, `org.scalatest.run`, является полным именем приложения. Scala запустит это приложение и передаст ему все остальные элементы командной строки в качестве аргументов. Аргумент `TVSetSpec` указывает на выполняемый набор. Результат показан на рис. 14.1.

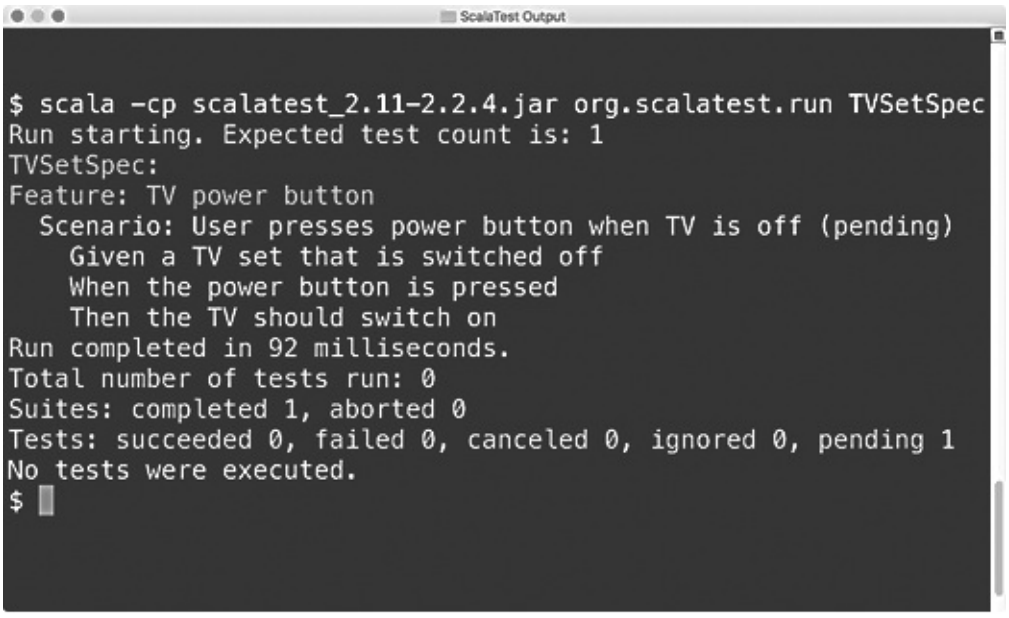


Рис. 14.1. Вывод, полученный при запуске org.scalatest.run

Резюме

В этой главе были показаны примеры подмешивания утверждений непосредственно в рабочий код, а также способы их внешней записи в тестах. Вы увидели, что программисты, работающие на Scala, могут воспользоваться преимуществами таких популярных средств тестирования от сообщества Java, как JUnit и TestNG, а также более новыми средствами, разработанными исключительно для Scala, такими как ScalaTest, ScalaCheck и specs2. И утверждения, встроенные в код, и внешние тесты могут помочь повысить качество ваших программных продуктов. Понимая важность этих технологий, мы представили их в данной главе, немного отклонившись от учебного материала по Scala. А в следующей главе вернемся к учебнику по языку и рассмотрим весьма полезное средство Scala — поиск по шаблону.

90 Метод assert определен в синглтон-объекте Predef, элементы которого автоматически импортируются в каждый исходный файл программы на языке Scala.

91 Более подробную информацию о ScalaTest можно найти на сайте http://www.scalatest.org/.

92 Программное средство specs2 можно загрузить с сайта http://specs2.org/.

15. Case-классы и поиск по шаблону

В этой главе вводятся понятия *case-классов* и *поиска по шаблону* (pattern matching) — парной конструкции, способствующей созданию обычных, неинкапсулированных структур данных. Особая польза от применения этих двух структур проявляется при работе с древовидными данными, подвергаемыми рекурсивной обработке.

Если вам уже приходилось программировать на функциональном языке, то, возможно, с поиском по шаблону вы уже знакомы. А вот case-классы будут для вас новым понятием. Case-классы в Scala являются способом применения к объектам поиска по шаблону, не требующим большого объема шаблонного кода. По сути, чтобы приспособить отдельно взятый класс к поиску по шаблону, нужно лишь добавить к нему ключевое слово `case`.

Эта глава начинается с простого примера case-классов и поиска по шаблону. Затем в ней дается обзор всех видов поддерживаемых шаблонов, рассматриваются роль запечатанных классов (sealed classes), тип `Option` и указываются некоторые неочевидные места в языке, где используется поиск по шаблону. В заключение представлен более объемный и приближенный к реальному использованию пример поиска по шаблону.

15.1. Простой пример

Прежде чем вникать во все правила и нюансы поиска по шаблону, есть смысл рассмотреть простой пример, дающий общее представление. Предположим, что нужно написать библиотеку, работающую с арифметическими выражениями и являющуюся, возможно, частью разрабатываемого предметно-ориентированного языка.

Первым шагом к решению этой задачи будет определение

вводимых данных. Чтобы ничего не усложнять, сконцентрируемся на арифметических выражениях, состоящих из переменных, чисел и унарных и бинарных операций. Все это выражается иерархией классов Scala, показанной в листинге 15.1.

Листинг 15.1. Определение case-классов

```
abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr)
extends Expr
case class BinOp(operator: String,
    left: Expr, right: Expr) extends Expr
```

Эта иерархия включает абстрактный базовый класс `Expr` с четырьмя подклассами, по одному для каждого вида рассматриваемых выражений[93]. Тела всех пяти классов пусты. Как уже упоминалось, при желании в Scala пустые тела классов в фигурные скобки можно не заключать, следовательно, определение `class C` аналогично определению `class C {}`.

Case-классы

Еще одной заслуживающей внимания особенностью объявлений в листинге 15.1 является то, что у каждого подкласса имеется модификатор `case`. Классы с таким модификатором называются case-классами. Использование этого модификатора заставляет компилятор Scala добавлять к вашему классу некоторые синтаксические возможности.

Первое синтаксическое удобство заключается в том, что к классу добавляется фабричный метод с именем этого класса. Это означает, к примеру, что для создания `var`-объекта можно

воспользоваться кодом `Var("x")`, не используя несколько более длинный вариант `new Var("x")`:

```
scala> val v = Var("x")
v: Var = Var(x)
```

Особую пользу от применения фабричных методов обеспечивает их вложенность. Поскольку теперь код не загромождён ключевыми словами `new`, структуру выражения можно воспринять с одного взгляда:

```
scala> val op = BinOp("+", Number(1), v)
op: BinOp = BinOp(+,Number(1.0),Var(x))
```

Второе синтаксическое удобство заключается в том, что все аргументы в списке параметров case-класса автоматически получают префикс `val`, то есть сохраняются в качестве полей:

```
scala> v.name
res0: String = x

scala> op.left
res1: Expr = Number(1.0)
```

Третье удобство состоит в том, что компилятор добавляет к вашему классу естественную реализацию методов `toString`, `hashCode` и `equals`. Они будут заниматься подготовкой данных к выводу, их хешированием и сравнением всего дерева, состоящего из класса, и (рекурсивно) всех его аргументов. Поскольку метод `==` в Scala передает полномочия методу `equals`, это означает, что элементы case-классов всегда сравниваются структурно:

```
scala> println(op)
BinOp(+,Number(1.0),Var(x))

scala> op.right == Var("x")
```

```
res3: Boolean = true
```

И наконец, для создания измененных копий компилятор добавляет к вашему классу метод `copy`. Этот метод пригодится для разработки нового экземпляра класса, аналогичного другому экземпляру, за исключением того, что он будет отличаться одним или двумя атрибутами. Метод работает за счет использования именованных параметров и параметров по умолчанию (см. раздел 8.8). Путем использования именованных параметров указываются требуемые изменения. А для любого неуказанного параметра задействуется значение из старого объекта. Посмотрим в качестве примера, как можно создать операцию, похожую на `op` во всем, кроме того, что будет изменен параметр `operator`:

```
scala> op.copy(operator = "-")
res4: BinOp = BinOp(-,Number(1.0),Var(x))
```

Все эти соглашения в качестве небольшого бонуса придают вашей работе массу удобств. Нужно просто указать модификатор `case`, и ваши классы и объекты приобретут гораздо более оснащенный вид. Они станут больше за счет создания дополнительных методов и подразумеваемого добавления поля для каждого параметра конструктора. Но самым большим преимуществом case-классов является поддержка ими поиска по шаблону.

Поиск по шаблону

Предположим, что нужно упростить арифметические выражения только что представленных видов. Существует множество возможных правил упрощения. В качестве иллюстрации можно показать три следующих правила:

```
UnOp("-", UnOp("-", e))  => e // Двойное отрицание
BinOp("+", e, Number(0)) => e // Прибавление нуля
```

```
BinOp("*", e, Number(1)) => e // Умножение на единицу
```

Как показано в листинге 15.2, чтобы в Scala сформировать ядро функции упрощения с использованием поиска по шаблону, эти правила можно взять практически в неизменном виде. Показанная в листинге функция `simplifyTop` может быть использована следующим образом:

```
scala> simplifyTop(UnOp("-", UnOp("-", Var("x"))))
res4: Expr = Var(x)
```

Листинг 15.2. Функция simplifyTop, выполняющая поиск по шаблону

```
  def simplifyTop(expr: Expr): Expr = expr match {
     case UnOp("-", UnOp("-", e))  => e // Двойное отрицание
        case BinOp("+", e, Number(0)) => e // Прибавление нуля
        case BinOp("*", e, Number(1)) => e // Умножение на единицу
    case _ => expr
  }
```

Правая часть `simplifyTop` состоит из выражения `match`, которое соответствует `switch` в Java, но записывается после выражения выбора. Иными словами, оно выглядит как:

```
выбор match { варианты }
```

вместо:

```
switch (выбор) { варианты }
```

Поиск по шаблону включает последовательность вариантов, каждый из которых начинается с ключевого слова `case`. Каждый вариант включает шаблон и одно или несколько выражений, которые будут вычислены при соответствии шаблону. Обозначение стрелки => отделяет шаблон от выражений.

Выражение `match` вычисляется проверкой соответствия каждого из шаблонов в порядке их написания. Выбирается первый же соответствующий шаблон, а также выбирается и выполняется та часть, которая следует за обозначением стрелки.

Шаблон-константа вида `+` или `1` соответствует значениям, равным константе в случае применения метода `==`. *Шаблон-переменная* вида `e` соответствует любому значению. Затем переменная ссылается на это же значение в правой части условия `case`. Обратите внимание на то, что в этом примере первые три варианта вычисляются в `e`, то есть в переменную, связанную внутри соответствующего шаблона. *Универсальный шаблон сопоставления* (`_`) также соответствует любому значению, но без представления имени переменной для ссылки на это значение. Посмотрите, как в листинге 15.2 выражение `match` заканчивается условием `case`, которое применяется в случае отсутствия соответствующих шаблонов и не предполагает никаких действий с выражением. Вместо этого получается просто выражение `expr`, в отношении которого и выполняется поиск по шаблону.

Шаблон-конструктор выглядит как `UnOp("-", e)`. Он отвечает всем значениям типа `UnOp`, первый аргумент которых соответствует `"-"`, а второй — `e`. Обратите внимание на то, что аргументы конструктора сами являются шаблонами. Это позволяет составлять многоуровневые шаблоны с использованием краткой формы записи.

Примером может послужить следующий шаблон:

```
UnOp("-", UnOp("-", e))
```

Представьте, что попытка реализации такой же функциональной возможности осуществляется с использованием шаблона проектирования visitor[94]! Практически так же трудно представить себе реализацию такой же функциональной возможности в виде длинной последовательности инструкций, проверок соответствия типам и явного приведения типов.

Сравнение match со switch

Выражения `match` могут быть представлены в качестве общих случаев `switch`-выражений в стиле Java. В Java `switch`-выражение может быть вполне естественно представлено в виде `match`-выражения, где каждый шаблон является константой, а последний шаблон может быть универсальным шаблоном сопоставления (который представлен в `switch`-выражении вариантом, используемым в случае отсутствия других соответствий).

И тем не менее следует учитывать три различия. Во-первых, `match` является выражением языка Scala, то есть оно всегда вычисляется в значение. Во-вторых, применяемые в Scala выражения вариантов никогда не «проваливаются» в следующий вариант. В-третьих, если не найдено соответствие ни одному из шаблонов, выдается исключение по имени `MatchError`. Следовательно, вам придется всегда обеспечивать охват всех возможных вариантов, даже если это будет означать указание варианта, применяемого при отсутствии всех ранее обозначенных соответствий и не предусматривающего никаких действий.

Пример показан в листинге 15.3. Второй вариант необходим, поскольку без него выражение `match` выдаст исключение `MatchError` для любого `expr`-аргумента, не являющегося `BinOp`. В данном примере для этого второго варианта не указано никакого кода, поэтому, если он сработает, ничего не произойдет. Результатом в любом случае станет `unit`-значение `()`, которое также будет результатом вычисления всего выражения `match`.

Листинг 15.3. Поиск по шаблону с пустым вариантом несоответствия

```
expr match {
  case BinOp(op, left, right) =>
    println(expr + " является бинарной операцией")
  case _ =>
}
```

15.2. Разновидности шаблонов

В предыдущем примере был дан краткий обзор некоторых разновидностей шаблонов. А теперь давайте потратим немного времени на более подробное изучение каждого из них.

Синтаксис шаблонов довольно прост, поэтому из-за него особо переживать не стоит. Все шаблоны выглядят точно так же, как и соответствующие им выражения. Например, если взять иерархию из листинга 15.1, то шаблон `Var(x)` соответствует любому выражению, содержащему переменную, с привязкой `x` к имени переменной. Будучи использованным в качестве выражения, `Var(x)` с точно таким же синтаксисом воссоздает эквивалентный объект, предполагая, что идентификатор `x` уже привязан к имени переменной. Поскольку в синтаксисе шаблонов все понятно, главное, на что следует обратить внимание, — то, какого вида шаблоны можно применять.

Универсальные шаблоны сопоставления

Универсальный шаблон сопоставления (`_`) соответствует абсолютно любому объекту. Вы уже видели, как он используется в качестве общего шаблона, выявляющего все оставшиеся варианты:

```
expr match {
  case BinOp(op, left, right) =>
```

```
    println(expr + " is a binary operation")
  case _ => // обработка общего варианта
}
```

Универсальные шаблоны сопоставления могут использоваться также для игнорирования тех частей объекта, которые не представляют для вас интереса. Например, в предыдущем примере нас не интересует, что представляют собой элементы бинарной операции, в нем лишь проверяется, является эта операция бинарной или нет. Поэтому, как показано в листинге 15.4, для элементов `BinOp` в коде также могут использоваться универсальные шаблоны сопоставления.

Листинг 15.4. Поиск по шаблону с универсальными шаблонами сопоставления

```
expr match {
   case BinOp(_, _, _) => println(expr + " is a
binary operation")
  case _ => println("It's something else")
}
```

Шаблоны-константы

Шаблон-константа соответствует только самому себе. В качестве константы может использоваться любой литерал. Например, шаблонами-константами являются `5`, `true` и `"hello"`. В качестве константы может использоваться также любой `val`- или синглтон-объект. Например, синглтон-объект `Nil` является шаблоном, соответствующим только пустому списку. Некоторые примеры шаблонов-констант показаны в листинге 15.5.

Листинг 15.5. Поиск по шаблону с использованием шаблонов-

констант

```
def describe(x: Any) = x match {
  case 5 => "five"
  case true => "truth"
  case "hello" => "hi!"
  case Nil => "the empty list"
  case _ => "something else"
}
```

А вот как поиск по шаблону, показанный в листинге 15.5, выглядит в действии:

```
scala> describe(5)
res6: String = five

scala> describe(true)
res7: String = truth

scala> describe("hello")
res8: String = hi!

scala> describe(Nil)
res9: String = the empty list

scala> describe(List(1,2,3))
res10: String = something else
```

Шаблоны-переменные

Шаблон-переменная соответствует любому объекту точно так же, как универсальный шаблон сопоставления, но, в отличие от него, Scala привязывает переменную к тому, что собой представляет объект. Затем эту переменную можно использовать для

дальнейшего воздействия на объект. Например, в листинге 15.6 показан поиск по шаблону, имеющий специальный вариант для нуля и общий вариант для всех остальных значений. В общем варианте используется шаблон-переменная, поэтому у него есть имя для переменной независимо от того, что это на самом деле.

Листинг 15.6. Поиск по шаблону с использованием шаблона-переменной

```
expr match {
  case 0 => "zero"
     case somethingElse => "not zero: " +
somethingElse
}
```

Переменная или константа? У шаблонов-констант могут быть условные наименования. Вы уже это видели, когда в качестве шаблона использовалось условное наименование `Nil`. А вот похожий пример, где в поиске по шаблону задействуются константы `E` (2,71828...) и `Pi` (3,14159...):

```
scala> import math.{E, Pi}
import math.{E, Pi}

scala> E match {
         case Pi => "strange math? Pi = " + Pi
         case _ => "OK"
       }
res11: String = OK
```

Как и ожидалось, значение `E` не равно значению `Pi`, поэтому вариант «математический казус» не используется.

А как компилятор Scala распознает `Pi` в качестве константы,

импортированной из `scala.math`, и не путает этот идентификатор с переменной, имеющей выбираемое значение? Чтобы не возникало путаницы, в Scala действует простое лексическое правило: обычное имя, начинающееся с буквы в нижнем регистре, считается переменной шаблона, а все другие ссылки считаются константами. Чтобы заметить разницу, создайте для `pi` псевдоним с первой буквой, указанной в нижнем регистре, и попробуйте в работе следующий код:

```
scala> val pi = math.Pi
pi: Double = 3.141592653589793

scala> E match {
         case pi => "strange math? Pi = " + pi
       }
res12: String = strange math? Pi = 2.718281828459045
```

Здесь компилятор даже не позволит вам добавить общий вариант. Поскольку `pi` является шаблоном-переменной, он будет соответствовать всем вводимым данным, поэтому до следующих вариантов дело просто не дойдет:

```
scala> E match {
         case pi => "strange math? Pi = " + pi
         case _ => "OK"
       }
<console>:12: warning: unreachable code
                case _ => "OK"
                          ^
```

Но при необходимости для шаблона-константы можно использовать имя, начинающееся с буквы в нижнем регистре, для чего придется воспользоваться одним из двух приемов. Если

константа является полем какого-нибудь объекта, перед ней можно поставить префикс-классификатор. Например, `pi` является шаблоном-переменной, а `this.pi` или `obj.pi` — константами, несмотря на то что их имена начинаются с букв в нижнем регистре. Если это не сработает (поскольку, к примеру, `pi` является локальной переменной), то, как вариант, можно будет заключить имя переменной в обратные кавычки. Например, `` `pi` `` будет опять восприниматься как константа, а не как переменная:

```
scala> E match {
         case `pi` => "strange math? Pi = " + pi
         case _ => "OK"
       }
res14: String = OK
```

Как вы, наверное, заметили, использование для идентификаторов в Scala синтаксиса с обратными кавычками во избежание в коде необычных обстоятельств преследует две цели. Здесь показано, что этот синтаксис может применяться для рассмотрения идентификатора с именем, начинающимся с буквы в нижнем регистре, в качестве константы при поиске по шаблону. Ранее, в разделе 6.10, было показано, что этот синтаксис может использоваться также для трактовки ключевого слова в качестве обычного идентификатора. Например, в выражении `writingThread.` `` `yield` `` `()` слово `yield` задействуется как идентификатор, а не как ключевое слово.

Шаблоны-конструкторы

Реальная эффективность поиска по шаблону проявляется именно в конструкторах. Шаблон-конструктор выглядит как `BinOp("+", e, Number(0))`. Он состоит из имени (`BinOp`), после которого в круглых скобках стоят несколько шаблонов: `"+"`, `e` и `Number(0)`. При условии, что имя обозначает case-класс, такой шаблон

показывает, что сначала выполняется проверка принадлежности элемента к названному case-классу, а затем проверка того, что параметры конструктора объекта соответствуют предоставленным дополнительным шаблонам.

Эти дополнительные шаблоны означают, что в шаблонах Scala поддерживается *поиск глубоких соответствий* (deep matches). В таких шаблонах проверяется не только предоставленный объект верхнего уровня, но и содержимое объекта на соответствие следующим шаблонам. Поскольку дополнительные шаблоны сами по себе могут быть шаблонами-конструкторами, их можно использовать для проверки объекта на произвольную глубину. Например, шаблон, показанный в листинге 15.7, проверяет, что объект верхнего уровня относится к типу `BinOp`, третьим параметром его конструктора является число `Number` и значение поля этого числа — `0`. Весь шаблон умещается в одну строку кода, хотя выполняет проверку на глубину в три уровня.

Листинг 15.7. Поиск по шаблону с использованием шаблона-конструктора

```
expr match {
  case BinOp("+", e, Number(0)) => println("a deep
match")
  case _ =>
}
```

Шаблоны-последовательности

По аналогии с поиском соответствий case-классам можно искать соответствия таким типам последовательностей, как `List` или `Array`. Можно воспользоваться тем же синтаксисом, но теперь в шаблоне разрешено указать любое количество элементов. В листинге 15.8 показан шаблон для проверки того факта, что

трехэлементный список начинается с нуля.

Листинг 15.8. Шаблон-последовательность фиксированной длины

```
expr match {
  case List(0, _, _) => println("found it")
  case _ =>
}
```

Если нужно определить соответствие последовательности без указания ее длины, то в качестве последнего элемента шаблона-последовательности можно указать шаблон `_*`. Он имеет весьма забавный вид и отвечает любому количеству элементов внутри последовательности, включая нуль элементов. В листинге 15.9 показан пример, соответствующий любому списку, который начинается с нуля, независимо от длины этого списка.

Листинг 15.9. Шаблон-последовательность произвольной длины

```
expr match {
  case List(0, _*) => println("found it")
  case _ =>
}
```

Шаблоны-кортежи

Можно также выполнять поиск соответствий кортежам. Шаблон вида `(a, b, c)` соответствует произвольному трехэлементному кортежу. Пример показан в листинге 15.10.

Листинг 15.10. Поиск по шаблону с использованием шаблона-кортежа

```
def tupleDemo(expr: Any) =
  expr match {
    case (a, b, c) => println("matched " + a + b +
c)
    case _ =>
  }
```

Если загрузить показанный в листинге 15.10 метод `tupleDemo` в интерпретатор и передать ему кортеж из трех элементов, получится следующая картина:

```
scala> tupleDemo(("a ", 3, "-tuple"))
matched a 3-tuple
```

Типизированные шаблоны

Типизированный шаблон (typed pattern) можно использовать в качестве удобного заменителя для проверок типов и приведения типов. Пример показан в листинге 15.11.

Листинг 15.11. Поиск по шаблону с использованием типизированных шаблонов

```
def generalSize(x: Any) = x match {
  case s: String => s.length
  case m: Map[_, _] => m.size
  case _ => -1
}
```

А вот несколько примеров применения `generalSize` в интерпретаторе Scala:

```
scala> generalSize("abc")
res16: Int = 3
```

```
scala> generalSize(Map(1 -> 'a', 2 -> 'b'))
res17: Int = 2

scala> generalSize(math.Pi)
res18: Int = -1
```

Метод `generalSize` возвращает размер или длину объектов различных типов. Типом его аргумента является `Any`, поэтому им может быть любое значение. Если типом аргумента является `String`, метод возвращает длину строки. Шаблон `s: String` является типизированным шаблоном и соответствует каждому (ненулевому) экземпляру класса `String`. Затем на эту строку ссылается шаблон-переменная `s`.

Заметьте: даже при том что `s` и `x` ссылаются на одно и то же значение, типом `x` является `Any`, а типом `s` — `String`. Поэтому в альтернативном выражении, соответствующем шаблону, можно воспользоваться кодом `s.length`, но нельзя — кодом `x.length`, поскольку в типе `Any` отсутствует компонент `length`. Эквивалентным, но более многословным способом достижения такого же эффекта соответствия типизированному шаблону является проверка типа с последующим приведением типа. В Scala для этого используется не такой синтаксис, как в Java. К примеру, для проверки того, относится ли выражение `expr` к типу `String`, применяется следующий код:

```
expr.isInstanceOf[String]
```

Для приведения того же выражения к типу `String` используется код:

```
expr.asInstanceOf[String]
```

Применяя проверку и приведение типа, можно переписать первый вариант предыдущего выражения соответствия, получив код, показанный в листинге 15.12.

Листинг 15.12. Использование isInstanceOf и asInstanceOf (неэффективный стиль)

```
if (x.isInstanceOf[String]) {
  val s = x.asInstanceOf[String]
  s.length
} else ...
```

Операторы `isInstanceOf` и `asInstanceOf` считаются предопределенными методами класса `Any`, получающими параметр типа в квадратных скобках. Фактически `x.asInstanceOf[String]` является частным случаем вызова метода с явно заданным параметром типа `String`.

Как вы уже заметили, написание проверок и приведений типов в Scala страдает излишним многословием. Сделано это намеренно, поскольку подобная практика не приветствуется. Как правило, лучше будет воспользоваться поиском по типизированному шаблону. В частности, такой подход будет оправдан, если нужно выполнить две операции — проверку типа и его приведение, поскольку обе они будут сведены к единственному поиску по шаблону.

Второй вариант выражения соответствия в листинге 15.11 содержит типизированный шаблон `m: Map[_, _]`. Он отвечает любому значению, являющемуся отображением каких-либо произвольных типов ключа и значения, и позволяет `m` ссылаться на это значение. Поэтому `m.size` имеет правильную типизацию и возвращает размер отображения. Знаки подчеркивания в типизированном шаблоне[95] подобны таким же знакам в универсальных шаблонах сопоставления. Вместо них можно указывать переменные типа с символами в нижнем регистре.

Затирание типов

А можно ли также проводить проверку на отображение с конкретными типами элементов? Это пригодилось бы, скажем, для проверки того, является ли заданное значение отображением типа `Int` на тип `Int`. Давайте попробуем:

```
scala> def isIntIntMap(x: Any) = x match {
         case m: Map[Int, Int] => true
         case _ => false
       }
<console>:9: warning: non-variable type argument Int in type
pattern    scala.collection.immutable.Map[Int,Int] (the
underlying of Map[Int,Int]) is unchecked since it is
eliminated by erasure
         case m: Map[Int, Int] => true
                 ^
```

В Scala точно так же, как и в Java, используется модель *затирания* обобщенных типов. Это означает, что в ходе выполнения программы никакая информация об аргументах типов не сохраняется. Следовательно, способов определения в ходе выполнения программы того, создавался ли заданный `Map`-объект с двумя `Int`-аргументами, а не с аргументами других типов, не существует. Система может лишь определить, что значение является отображением (`Map`) каких-то произвольных параметров типа. Убедиться в таком поведении можно, применив `isIntIntMap` к различным экземплярам класса `Map`:

```
scala> isIntIntMap(Map(1 -> 1))
res19: Boolean = true

scala> isIntIntMap(Map("abc" -> "abc"))
```

```
res20: Boolean = true
```

Первое применение возвращает `true`, что выглядит вполне корректно, но второе применение также возвращает `true`, что может оказаться сюрпризом. Чтобы оповестить вас о возможном непонятном поведении программы в ходе ее выполнения, компилятор выдает предупреждение о том, что он это поведение не контролирует, похожее на те, что показаны ранее.

Единственным исключением из правила затирания являются массивы, поскольку в Java, а также в Scala они обрабатываются особым образом. Тип элемента массива сохраняется вместе со значением массива, поэтому к нему можно применить поиск по шаблону. Пример выглядит следующим образом:

```
scala> def isStringArray(x: Any) = x match {
         case a: Array[String] => "yes"
         case _ => "no"
}
isStringArray: (x: Any)String

scala> val as = Array("abc")
as: Array[String] = Array(abc)
scala> isStringArray(as)
res21: String = yes

scala> val ai = Array(1, 2, 3)
ai: Array[Int] = Array(1, 2, 3)

scala> isStringArray(ai)
res22: String = no
```

Привязка переменной

Кроме использования отдельного взятого шаблона-переменной

можно также добавить переменную к любому другому шаблону. Нужно просто указать имя переменной, знак «эт» (@), а затем шаблон. Это даст вам шаблон с привязанной переменной, то есть шаблон для выполнения обычного поиска по шаблону с возможностью в случае соответствия присвоить переменной конкретный объект, как и при использовании обычного шаблона-переменной.

В качестве примера в листинге 15.13 показан поиск по шаблону — поиск операции получения абсолютного значения, применяемой в строке дважды. Такое выражение может быть упрощено путем однократного присвоения абсолютного значения.

Листинг 15.13. Шаблон с привязанной переменной (посредством использования знака @)

```
expr match {
  case UnOp("abs", e @ UnOp("abs", _)) => e   case
_ =>
}
```

Пример, показанный в листинге 15.13, включает шаблон с привязкой переменной, где в качестве переменной выступает `e`, а в качестве шаблона — `UnOp("abs", _)`. Если будет найдено соответствие всему шаблону, то та часть, которая отвечает `UnOp("abs", _)`, станет доступна как значение переменной `e`. Результатом варианта будет просто `e`, поскольку у `e` такое же значение, что и у `expr`, но с меньшим на единицу количеством операций получения абсолютного значения.

15.3. Ограничители шаблонов

Иногда синтаксический поиск по шаблонам недостаточно точен. Предположим, к примеру, что перед вами стоит задача

сформулировать правило упрощения, заменяющее выражение сложения с двумя одинаковыми операндами, такое как `e + e`, умножением на два, например `e * 2`. На языке деревьев `Expr` выражение вида:

```
BinOp("+", Var("x"), Var("x"))
```

этим правилом будет превращено в:

```
BinOp("*", Var("x"), Number(2))
```

Правило можно попробовать выразить следующим образом:

```
scala> def simplifyAdd(e: Expr) = e match {
           case BinOp("+", x, x) => BinOp("*", x,
Number(2))
         case _ => e        }
<console>:14: error: x is already defined as value
x
           case BinOp("+", x, x) => BinOp("*", x,
Number(2))
                                ^
```

Попытка будет неудачной, поскольку в Scala не допускается разветвленности шаблонов: шаблон-переменная может появляться в шаблоне только один раз. Но, как показано в листинге 15.14, соответствие можно переформулировать с помощью *ограничителя шаблонов* (pattern guard).

Листинг 15.14. Поиск по шаблону с применением ограничителя шаблонов

```
scala> def simplifyAdd(e: Expr) = e match {
         case BinOp("+", x, y) if x == y =>
           BinOp("*", x, Number(2))
```

```
        case _ => e          }
simplifyAdd: (e: Expr)Expr
```

Ограничитель шаблона указывается после шаблона и начинается с ключевого слова `if`. В качестве ограничителя может использоваться произвольное булево выражение, которое обычно ссылается на переменные в шаблоне. При наличии ограничителя шаблонов соответствие считается найденным, только если ограничитель вычисляется в `true`. Таким образом, первый вариант показанного ранее кода соответствует только тем бинарным операциям, у которых два одинаковых операнда.

А вот как выглядят некоторые другие ограниченные шаблоны:

```
// соответствует только положительным целым числам
case n: Int if 0 < n => ...

// соответствует только строкам, начинающимся с буквы 'a'
case s: String if s(0) == 'a' => ...
```

15.4. Наложение шаблонов

Шаблоны применяются в том порядке, в котором они указаны. Версия упрощения, показанная в листинге 15.15, представляет собой пример, в котором порядок следования вариантов имеет значение.

Листинг 15.15. Выражение соответствия, где порядок следования вариантов имеет значение

```
def simplifyAll(expr: Expr): Expr = expr match {
  case UnOp("-", UnOp("-", e)) =>
         simplifyAll(e)  //  '-'  является  своей
```

```
собственной обратной величиной
  case BinOp("+", e, Number(0)) =>
     simplifyAll(e) // '0' нейтральный элемент для
'+'
  case BinOp("*", e, Number(1)) =>
     simplifyAll(e) // '1' нейтральный элемент для
'*'
  case UnOp(op, e) =>
    UnOp(op, simplifyAll(e))
  case BinOp(op, l, r) =>
    BinOp(op, simplifyAll(l), simplifyAll(r))
  case _ => expr
}
```

Версия упрощения, показанная в листинге 15.15, станет применять правила упрощения в любом месте выражения, а не только в его верхней части, как это сделала бы версия `simplifyTop`. Эту версию можно вывести из версии `simplifyTop` путем добавления двух дополнительных вариантов для обычных унарных и бинарных выражений (варианты четыре и пять в листинге 15.15).

В четвертом варианте используется шаблон `UnOp(op, e)`, который соответствует любой унарной операции. Оператор и операнд унарной операции могут быть какими угодно. Они привязаны к шаблонам-переменным `op` и `e` соответственно. Альтернативой в данном варианте будет рекурсивное применение `simplifyAll` к операнду `e` с последующим перестроением той же самой унарной операции с (возможно) упрощенным операндом. Пятый вариант для `BinOp` аналогичен четвертому: он является вариантом всеобщего отлова для произвольных бинарных операций, который рекурсивно применяет метод упрощения к своим двум операндам.

Важным обстоятельством в этом примере является то, что

варианты всеобщего отлова следуют после более конкретизированных правил упрощения. Если расположить их в другом порядке, то вариант всеобщего отлова будет запущен вместо более конкретизированных правил. Во многих случаях компилятор станет жаловаться на такие попытки. Например, вот как выглядит выражение соответствия, которое не пройдет компиляцию, поскольку первый вариант будет соответствовать всему тому, чему будет соответствовать второй вариант:

```
scala> def simplifyBad(expr: Expr): Expr = expr
match {
                  case UnOp(op, e) => UnOp(op,
simplifyBad(e))
         case UnOp("-", UnOp("-", e)) => e
       }
<console>:21: warning: unreachable code
         case UnOp("-", UnOp("-", e)) => e
                                         ^
```

15.5. Запечатанные классы

При написании поиска по шаблонам нужно удостовериться в том, что охвачены все возможные варианты. Иногда это можно сделать, добавив в конец проверки соответствия общий вариант, но этот способ применим только тогда, когда есть вполне определенное общее поведение. А что делать, если его нет? Как узнать, что охвачены все варианты и нет опасности что-либо упустить?

Чтобы определить пропущенные в выражении `match` комбинаций шаблонов, можно обратиться за помощью к компилятору Scala. Для этого у компилятора должна быть возможность сообщить обо всех возможных вариантах. По сути, в Scala это сделать невозможно, поскольку классы могут быть определены в любое время и в произвольных блоках компиляции. Например, ничто не помешает вам добавить к иерархии класса

Expr пятый case-класс не в том блоке компиляции, где определены четыре других case-класса, а в другом.

Альтернативой этому может стать превращение родительского класса ваших case-классов в *запечатанный* класс. У такого запечатанного класса не может быть никаких добавленных подклассов, кроме тех, которые определены в том же самом файле. Особую пользу из этого можно извлечь при поиске по шаблону, поскольку запечатанность класса будет означать, что беспокоиться придется только по поводу тех подклассов, о которых вам уже известно. Более того, будет улучшена поддержка со стороны компилятора. При поиске соответствий case-классам, являющимся наследниками запечатанного класса, компилятор в предупреждении отметит пропущенные комбинации шаблонов.

Если создается иерархия классов, предназначенная для поиска по шаблону, нужно предусмотреть ее запечатанность. Для этого просто поставьте перед классом на вершине иерархии ключевое слово sealed. Программисты, использующие вашу иерархию классов, при поиске по шаблону будут чувствовать себя уверенно. Таким образом, ключевое слово sealed зачастую выступает лицензией на поиск по шаблону. Пример, в котором Expr превращается в запечатанный класс, показан в листинге 15.16.

Листинг 15.16. Запечатанная иерархия case-классов

```
sealed abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr)
extends Expr
case class BinOp(operator: String,
    left: Expr, right: Expr) extends Expr
```

А теперь определим поиск по шаблону, где упущены некоторые

возможные варианты:

```
def describe(e: Expr): String = e match {
  case Number(_) => "a number"
  case Var(_) => "a variable"
}
```

В результате будет получено следующее предупреждение компилятора:

```
warning: match is not exhaustive!
missing combination            UnOp
missing combination           BinOp
```

Оно сообщает о существовании риска выдачи вашим кодом исключения `MatchError`, поскольку некоторые возможные шаблоны (`UnOp`, `BinOp`) не обрабатываются. Предупреждение указывает на потенциальный источник сбоя в ходе выполнения программы, поэтому обычно оно хорошо помогает при корректировке кода программы.

Но порой можно столкнуться с ситуацией, в которой компилятор при выдаче предупреждений проявляет излишнюю дотошность. Например, из контекста может быть известно, что показанный ранее метод `describe` будет применяться только к выражениям типа `Number` или `Var`, следовательно, исключение `MatchError` выдаваться не будет. Чтобы избавиться от предупреждения, к методу можно добавить третий, общий вариант:

```
def describe(e: Expr): String = e match {
  case Number(_) => "a number"
  case Var(_) => "a variable"
   case _ => throw new RuntimeException // Не
должно произойти
}
```

Решение вполне работоспособное, но не идеальное. Вряд ли вас обрадует принуждение к добавлению кода, который никогда не будет выполнен (по вашему мнению), лишь для того, чтобы успокоить компилятор.

Более экономной альтернативой станет добавление к селектору выражения поиска по шаблону аннотации @unchecked. Делается это следующим образом:

```
def describe(e: Expr): String = (e: @unchecked)
match {
  case Number(_) => "a number"
  case Var(_) => "a variable"
}
```

Аннотации рассматриваются в главе 27. В общем, аннотации можно добавлять к выражению точно так же, как это делается при добавлении типа: нужно после выражения поставить двоеточие, знак «эт» и указать название аннотации. Например, в данном случае к переменной e добавляется аннотация @unchecked, для чего используется код e: @unchecked. Аннотация @unchecked имеет особое значение для поиска по шаблону. Если выражение селектора поиска содержит данную аннотацию, исчерпывающая проверка последующих шаблонов будет подавлена.

15.6. Тип Option

Для значений, которые могут отсутствовать, в Scala имеется стандартный тип Option. Значение такого типа может быть двух видов: это либо Some(*x*), где *x* — реальное значение, либо None-объект, представляющий отсутствующее значение.

Значения, которые могут отсутствовать, выдаются некоторыми стандартными операциями, имеющимися в коллекции Scala. Например, метод get из Scala-класса Map производит

Some(значение), если найдено значение, соответствующее заданному ключу, или None, если заданный ключ не определен в Map-объекте. Пример выглядит следующим образом:

```
scala> val capitals =
             Map("France" -> "Paris", "Japan" ->
"Tokyo")
capitals:
scala.collection.immutable.Map[String,String] =
Map(France -> Paris, Japan -> Tokyo)
scala> capitals get "France"
res23: Option[String] = Some(Paris)

scala> capitals get "North Pole"
res24: Option[String] = None
```

Самый распространенный способ независимого получения значений, которые могут отсутствовать, заключается в использовании поиска по шаблону, например:

```
scala> def show(x: Option[String]) = x match {
         case Some(s) => s
         case None => "?"
}
show: (x: Option[String])String

scala> show(capitals get "Japan")
res25: String = Tokyo

scala> show(capitals get "France")
res26: String = Paris

scala> show(capitals get "North Pole")
```

```
res27: String = ?
```

Тип Option применяется в программах на языке Scala довольно часто. Его использование можно сравнить с доминирующей в Java идиомой показа отсутствия значения посредством null. Например, метод get из java.util.HashMap возвращает либо значение, сохраненное в HashMap, либо null, если значение не было найдено. В Java такой подход работает, но, применяя его, легко допустить ошибку, поскольку на практике довольно трудно отследить, каким переменным в программе разрешено иметь значение null.

Если переменной разрешено иметь значение null, вы должны вспомнить о проверке ее на наличие этого значения при каждом использовании. Если забыть выполнить эту проверку, появится вероятность выдачи в ходе выполнения программы исключений NullPointerException. Поскольку такие исключения могут выдаваться довольно редко, то с выявлением ошибки при тестировании могут возникнуть затруднения. В языке Scala такой подход вообще не сработает, поскольку в нем имеется возможность сохранения типов значений в хеш-отображениях, а null не является допустимым элементом для типов значений. Например, aHashMap[Int, Int] не может вернуть null, чтобы обозначить отсутствие элемента.

Вместо этого в Scala для указания значения, которое может отсутствовать, применяется тип Option. По сравнению с подходом, используемым в Java, у такого подхода к работе со значениями, которые могут отсутствовать, имеется ряд преимуществ. Во-первых, тем, кто читает код, намного понятнее, что переменная, чьим типом является Option[String], является переменной String, у которой может не быть значения, а не переменной типа String, которая порой может иметь значение null. Во-вторых, и это важнее всего, рассмотренные ранее ошибки программирования, связанные с использованием переменной, которая может иметь значение null, без предварительной

проверки ее на null превращаются в Scala в ошибку типа. Если переменная имеет тип Option[String], то при попытке ее использования в качестве строки ваша программа на Scala не пройдет компиляцию.

15.7. Повсеместное использование шаблонов

Шаблоны можно применять не только в отдельно взятых выражениях поиска соответствий, но и во многих других местах программы на языке Scala. Рассмотрим несколько подобных мест использования шаблонов.

Шаблоны в определениях переменных

При определении val- или var-переменной вместо простых идентификаторов разрешено использовать шаблоны. Например, можно, как показано в листинге 15.17, разобрать кортеж и присвоить каждую его часть своей собственной переменной.

Листинг 15.17. Определение нескольких переменных с помощью одного присваивания

```
scala> val myTuple = (123, "abc")
myTuple: (Int, String) = (123,abc)

scala> val (number, string) = myTuple
number: Int = 123
string: String = abc
```

Особую пользу из этой конструкции можно извлечь в ходе работы с case-классами. Если точно известен case-класс, с которым ведется работа, вы можете разобрать его с помощью шаблона. Пример выглядит следующим образом:

```
scala> val exp = new BinOp("*", Number(5), Number(1))
exp: BinOp = BinOp(*,Number(5.0),Number(1.0))

scala> val BinOp(op, left, right) = exp
op: String = *
left: Expr = Number(5.0)
right: Expr = Number(1.0)
```

Последовательности вариантов в качестве частично применяемых функций

Последовательность вариантов (то есть альтернатив), заключенную в фигурные скобки, можно задействовать везде, где может использоваться функциональный литерал. По сути, последовательность вариантов и есть функциональный литерал, только более универсальный. Вместо наличия единственной точки входа и списка параметров последовательность вариантов обладает несколькими точками входа, у каждой из которых собственный список параметров. Любой вариант является точкой входа в функцию, а параметры указываются с помощью шаблона. Телом каждой точки входа является правосторонняя часть варианта.

Простой пример выглядит следующим образом:

```
val withDefault: Option[Int] => Int = {
  case Some(x) => x   case None => 0
}
```

В теле этой функции имеются два варианта. Первый соответствует Some и возвращает число, находящееся внутри Some. Второй соответствует None и возвращает стандартное значение нуль. А вот как используется эта функция:

```
scala> withDefault(Some(10))
res28: Int = 10

scala> withDefault(None)
res29: Int = 0
```

Такая возможность особенно полезна для библиотеки акторов Akka, поскольку она позволяет определить ее метод `receive` в виде серии вариантов:

```
var sum = 0
def receive = {

case Data(byte) =>
  sum += byte

case GetChecksum(requester) =>
  val checksum = ~(sum & 0xFF) + 1
  requester ! checksum
}
```

Стоит также упомянуть еще одно общее правило: последовательность вариантов дает вам частично применяемую функцию. Если применить такую функцию в отношении не поддерживаемого ею значения, она выдаст исключение времени выполнения. Например, здесь показана частично применяемая функция, возвращающая второй элемент списка, состоящего из целых чисел:

```
val second: List[Int] => Int = {
  case x :: y :: _ => y }
```

При компиляции этого кода компилятор вполне резонно выведет предупреждение о том, что поиск соответствий не

охватывает все возможные варианты:

```
<console>:17: warning: match is not exhaustive!
missing combination Nil
```

Функция справится со своей задачей, если ей передать список, состоящий из трех элементов, но не станет работать при передаче пустого списка:

```
scala> second(List(5, 6, 7))
res24: Int = 6

scala> second(List())
scala.MatchError: List()
      at $anonfun$1.apply(<console>:17)
      at $anonfun$1.apply(<console>:17)
```

Если нужно проверить, определена ли частично применяемая функция, то сначала следует сообщить компилятору о том, что вы знаете, что работаете с частично применяемыми функциями. Тип `List[Int] => Int` включает все функции, получающие из целочисленных списков целочисленные значения независимо от того, частично они применяются или нет. Тип, который включает только частично применяемые функции, получающие из целочисленных списков целочисленные значения, записывается в виде `PartialFunction[List[Int],Int]`. Вот еще один вариант функции `second`, определенной с типом частично применяемой функции:

```
val second: PartialFunction[List[Int],Int] = {
  case x :: y :: _ => y }
```

У частично применяемых функций имеется метод `isDefinedAt`, который может использоваться для тестирования того, определена ли функция в отношении конкретного значения.

В данном случае функция определена для любого списка, у которого имеется по крайней мере два элемента:

```
scala> second.isDefinedAt(List(5,6,7))
res30: Boolean = true

scala> second.isDefinedAt(List())
res31: Boolean = false
```

Типичным примером частично применяемой функции может послужить функциональный литерал поиска по шаблону, подобный представленному в предыдущем примере. Фактически такое выражение преобразуется компилятором Scala в частично применяемую функцию путем двойного преобразования шаблонов: один раз для реализации реальной функции, а второй — для проверки того, определена функция или нет.

Например, функциональный литерал `{ case x :: y :: _ => y }` преобразуется в следующее значение частично применяемой функции:

```
new PartialFunction[List[Int], Int] {
  def apply(xs: List[Int]) = xs match {
    case x :: y :: _ => y
  }
  def isDefinedAt(xs: List[Int]) = xs match {
    case x :: y :: _ => true
    case _ => false
  }
}
```

Это преобразование осуществляется в том случае, когда объявляемым типом функционального литерала является `PartialFunction`. Если объявляемый тип просто `Function1` или не указан, функциональный литерал вместо этого преобразуется в

полноценную функцию.

Вообще-то полноценными функциями нужно пробовать пользоваться везде, где только можно, поскольку использование частично применяемых функций допускает возникновение ошибок времени выполнения, в устранении которых компилятор вам помочь не в силах. Но иногда частично применяемые функции приносят реальную пользу. Вам следует обеспечить, чтобы этим функциям не было предоставлено необрабатываемое значение. Как вариант, можно воспользоваться средой, ожидающей использования частично применяемых функций и поэтому всегда перед вызовом функции применяющей проверку, выполняемую функцией `isDefinedAt`. Последнее проиллюстрировано приведенным ранее примером реагирования, где аргументом выступает частично применяемая функция, определение которой дано в точности для тех сообщений, которые нужно обработать вызывающему коду.

Шаблоны в выражениях for

Шаблон, как показано в листинге 15.18, можно использовать также для выражения `for`. Это выражение извлекает все пары «ключ — значение» из отображения `capitals` (столицы). Каждая пара соответствует шаблону `(country, city)` (страна, город), который определяет две переменные, `country` и `city`.

Шаблон пар, показанный в листинге 15.18, был специально приспособлен, поскольку выявление соответствия ему никогда не даст сбой. Конечно, `capitals` выдает последовательность пар, таким образом, можно быть уверенным, что каждая сгенерированная пара будет соответствовать шаблону пар.

Листинг 15.18. Выражение for с шаблоном-кортежем

```
scala> for ((country, city) <- capitals)
```

```
            println("The capital of " + country + "
is " + city)
The capital of France is Paris
The capital of Japan is Tokyo
```

Но с равной долей вероятности возможно, что шаблон не будет соответствовать сгенерированному значению. Именно такой случай показан в листинге 15.19.

Листинг 15.19. Отбор элементов списка, соответствующих шаблону

```
scala> val results = List(Some("apple"), None,
           Some("orange"))
results: List[Option[String]] = List(Some(apple),
None,
Some(orange))

scala> for (Some(fruit) <- results) println(fruit)
apple
orange
```

В этом примере показано, что сгенерированные значения, не соответствующие шаблону, отбрасываются. Например, второй элемент `None` в получившемся списке не соответствует шаблону `Some(fruit)`, поэтому в выводимой информации его нет.

15.8. Большой пример

После изучения различных форм шаблонов может быть интересно посмотреть на их применение в более существенном примере. Предлагаемая задача заключается в написании форматирующего выражения класса, который выводит арифметическое выражение в двумерной разметке. Такое выражение деления, как `x / (x + 1)`,

должно быть выведено вертикально — с числителем, показанным над знаменателем:

```
x
-----
x + 1
```

В качестве еще одного примера здесь в двумерной разметке показано выражение `((a / (b * c) + 1 / n) / 3)`:

```
  a     1
----- + -
b * c   n
---------
3
```

Исходя из этих примеров, можно прийти к выводу, что манипулированием разметкой должен заняться класс — назовем его `ExprFormatter`, поэтому есть смысл воспользоваться библиотекой разметки, разработанной в главе 10. Мы также используем семейство case-классов `Expr`, ранее уже встречавшееся в этой главе, и поместим в именованные пакеты как библиотеку разметки из главы 10, так и средство форматирования выражений. Полный код этого примера будет показан в листингах 15.20 и 15.21.

Сначала полезно будет сосредоточиться на горизонтальной разметке. Структурированное выражение:

```
BinOp("+",
      BinOp("*",
            BinOp("+", Var("x"), Var("y")),
            Var("z")),
      Number(1))
```

должно привести к выводу `(x + y) * z + 1`. Обратите внимание на обязательность круглых скобок вокруг выражения `x + y` и их

необязательность вокруг выражения `(x + y) * z`. Чтобы разметка получилась максимально разборчивой, следует стремиться отказаться от избыточных круглых скобок и обеспечить наличие всех обязательных.

Чтобы узнать, куда ставить круглые скобки, коду необходимо быть в курсе относительной приоритетности каждого оператора, поэтому неплохо было бы сначала отрегулировать именно этот вопрос. Относительную приоритетность можно выразить непосредственно в виде литерала отображения следующей формы:

```
Map(
  "|" -> 0, "||" -> 0,
  "&" -> 1, "&&" -> 1, ...
)
```

Конечно, вам потребуется предварительно выполнить определенный объем вычислительной работы для назначения приоритетности. Удобнее будет просто определить группы операторов с нарастающей степенью приоритетности, а затем, исходя из этого, вычислить приоритетность каждого оператора. Соответствующий код показан в листинге 15.20.

Листинг 15.20. Верхняя половина средства форматирования выражений

```
package org.stairwaybook.expr
import org.stairwaybook.layout.Element.elem

sealed abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr)
extends Expr
```

```
case class BinOp(operator: String,
    left: Expr, right: Expr) extends Expr

class ExprFormatter {

   // Содержит операторы в группах с нарастающей
степенью приоритетности
  private val opGroups =
    Array(
      Set("|", "||"),
      Set("&", "&&"),
      Set("^"),
      Set("==", "!="),
      Set("<", "<=", ">", ">="),
      Set("+", "-"),
      Set("*", "%")
    )

     //  Отображение  операторов  на  их  степень
приоритетности
  private val precedence = {
    val assocs =
      for {
        i <- 0 until opGroups.length
        op <- opGroups(i)
      } yield op -> i     assocs.toMap
  }

  private val unaryPrecedence = opGroups.length
  private val fractionPrecedence = -1

  // продолжение в листинге 15.21...
```

Переменная `precedence` является отображением операторов на степень их приоритетности, представленную целыми числами, начинающимися с нуля. Приоритетность вычисляется с использованием выражения `for` с двумя генераторами. Первый генератор вырабатывает каждый индекс `i` массива `opGroups`. Второй генератор — каждый оператор `op`, находящийся в `opGroups(i)`. Для каждого такого оператора выражение `for` выдает привязку оператора `op` к его индексу `i`. В результате приоритетность оператора берется из его относительной позиции в массиве.

Привязки записываются с использованием инфиксной стрелки, например `op -> i`. До сих пор привязки были показаны только как часть конструкций отображений, но они имеют значение и сами по себе. Фактически привязка `op -> i` является не чем иным, как парой `(op, i)`.

Теперь, зафиксировав степень приоритетности всех бинарных операторов, за исключением `/`, есть смысл обобщить эту концепцию, охватив также унарные операторы. Степень приоритетности унарного оператора выше степени приоритетности любого бинарного оператора. Поэтому для переменной `unaryPrecedence`, показанной в листинге 15.20, устанавливается значение длины массива `opGroups` на единицу большее, чем степень приоритетности операторов `*` и `%`. Степень приоритетности оператора деления рассматривается не так, как этот параметр других операторов, поскольку для дробей используется вертикальная разметка. Но, конечно же, было бы удобно присвоить оператору деления специальное значение степени приоритетности `-1`, поэтому переменная `fractionPrecedence` будет инициализирована значением `-1`, как показано в листинге 15.20.

После такой подготовительной работы можно приступать к написанию основного метода `format`. Этот метод получает два аргумента: выражение `e`, имеющее тип `Expr`, и степень

приоритетности enclPrec того оператора, который непосредственно заключен в выражение e. (Если в выражении нет никакого оператора, значение enclPrec должно быть нулевым.) Метод выдает элемент разметки, представленный в виде двумерного массива символов. В листинге 15.21 показана остальная часть класса ExprFormatter, включающая три метода. Первый метод, stripDot, является вспомогательным. Далее идет закрытый метод format, выполняющий основную работу по форматированию выражений. Последний метод, который также называется format, представляет собой единственный открытый метод в библиотеке, который получает выражение для форматирования. Закрытый метод format проделывает свою работу, выполняя поиск по шаблону в отношении разновидностей выражения. У выражения поиска соответствия имеется пять вариантов, каждый из которых будет рассмотрен отдельно.

Листинг 15.21. Нижняя половина средства форматирования выражений

```
// ... продолжение, начало в листинге 15.20
import org.stairwaybook.layout.Element
private def format(e: Expr, enclPrec: Int):
Element =
  e match {

    case Var(name) =>
      elem(name)

    case Number(num) =>
      def stripDot(s: String) =
            if (s endsWith ".0") s.substring(0,
s.length - 2)
        else s
```

```
        elem(stripDot(num.toString))

      case UnOp(op, arg) =>
        elem(op) beside format(arg, unaryPrecedence)
      case BinOp("/", left, right) =>
        val top = format(left, fractionPrecedence)
        val bot = format(right, fractionPrecedence)
              val line = elem('-', top.width max
bot.width, 1)
        val frac = top above line above bot
        if (enclPrec != fractionPrecedence) frac
        else elem(" ") beside frac beside elem(" ")

      case BinOp(op, left, right) =>
        val opPrec = precedence(op)
        val l = format(left, opPrec)
        val r = format(right, opPrec + 1)
          val oper = l beside elem(" " + op + " ")
beside r
        if (enclPrec <= opPrec) oper
        else elem("(") beside oper beside elem(")")
    }

    def format(e: Expr): Element = format(e, 0)
  }
```

Первый вариант имеет следующий вид:

```
case Var(name) =>
  elem(name)
```

Если выражение является переменной, результатом станет элемент, сформированный из имени переменной.

Второй вариант выглядит следующим образом:

```
case Number(num) =>
  def stripDot(s: String) =
    if (s endsWith ".0") s.substring(0, s.length -
2)
    else s   elem(stripDot(num.toString))
```

Если выражение является числом, то результатом станет элемент, сформированный из значения числа. Функция `stripDot` очистит изображение числа с плавающей точкой, удалив из строки любой суффикс вида `".0"`.

А вот как выглядит третий вариант:

```
case UnOp(op, arg) =>
  elem(op) beside format(arg, unaryPrecedence)
```

Если выражение представляет собой унарную операцию `UnOp(op, arg)`, результат будет сформирован из операции `op` и результата форматирования аргумента `arg` с самой высокой из возможных степенью приоритетности, имеющейся в данной среде[96]. Это означает, что, если аргумент `arg` является бинарной операцией (но не делением), он всегда будет отображаться в круглых скобках.

Четвертый вариант представлен следующим кодом:

```
case BinOp("/", left, right) =>
  val top = format(left, fractionPrecedence)
  val bot = format(right, fractionPrecedence)
  val line = elem('-', top.width max bot.width, 1)
  val frac = top above line above bot
  if (enclPrec != fractionPrecedence) frac
  else elem(" ") beside frac beside elem(" ")
```

Если выражение имеет вид дроби, промежуточный результат `frac` формируется путем помещения отформатированных операндов `left` и `right` друг над другом с разделительным элементом в виде горизонтальной линии. Ширина горизонтальной линии равна максимальной ширине отформатированных операндов. Промежуточный результат становится окончательным результатом, если только дробь сама по себе не появляется в виде аргумента еще одной функции. В последнем случае по обе стороны `frac` добавляется по пробелу. Чтобы понять, зачем это делается, рассмотрим выражение `(a / b) / c`.

Без коррекции по ширине при форматировании этого выражения получится следующая картина:

```
a
-
b
-
c
```

Вполне очевидно наличие проблемы с разметкой: непонятно, где именно находится дробная черта верхнего уровня. Показанное ранее выражение может означать либо `(a / b) / c`, либо `a / (b / c)`. Чтобы устранить неоднозначность, с обеих сторон разметки вложенной дроби `a / b` нужно добавить пробелы. Тогда разметка станет понимаемой однозначно:

```
a
-
b
---
c
```

Пятый и последний вариант выглядит следующим образом:

```
case BinOp(op, left, right) =>
  val opPrec = precedence(op)
  val l = format(left, opPrec)
  val r = format(right, opPrec + 1)
   val oper = l beside elem(" " + op + " ") beside
r
  if (enclPrec <= opPrec) oper
  else elem("(") beside oper beside elem(")")
```

Этот вариант применяется ко всем остальным бинарным операциям. Поскольку он указан после варианта, который начинается со следующего кода:

```
case BinOp("/", left, right) => ...
```

понятно, что оператор `op` в шаблоне `BinOp(op, left, right)` не может быть оператором деления. Для форматирования такой бинарной операции нужно сначала отформатировать его операнды `left` и `right`. В качестве параметра степени приоритетности для форматирования левого операнда используется `opPrec` оператора `op`, а для правого операнда берется степень на единицу больше. Такая схема обеспечивает также правильное отображение ассоциативности, выраженное круглыми скобками.

Например, операция

```
BinOp("-",    Var("a"),    BinOp("-",    Var("b"),
Var("c")))
```

будет вполне корректно выражена с применением круглых скобок в виде `a - (b - c)`. Затем путем выстраивания в линию операндов `left` и `right`, разделенных оператором, формируется промежуточный результат `oper`. Если степень приоритетности текущего оператора ниже, чем степень приоритетности оператора, заключенного в скобки, `oper` помещается между круглыми скобками, в противном случае он возвращается в исходном виде.

На этом разработка закрытой функции `format` завершается. Остается только открытый метод `format`, позволяющий программистам, создающим клиентский код, форматировать высокоуровневые выражения без передачи аргумента, содержащего степень приоритетности. Демонстрационная программа, с помощью которой проверяется `ExprFormatter`, показана в листинге 15.22.

Листинг 15.22. Приложение, выполняющее вывод отформатированных выражений

```
import org.stairwaybook.expr._

object Express extends App {

  val f = new ExprFormatter

    val e1 = BinOp("*", BinOp("/", Number(1),
Number(2)),
                           BinOp("+", Var("x"),
Number(1)))

    val e2 = BinOp("+", BinOp("/", Var("x"),
Number(2)),
                        BinOp("/", Number(1.5),
Var("x")))

  val e3 = BinOp("/", e1, e2)

  def show(e: Expr) = println(f.format(e)+ "\n\n")

  for (e <- Array(e1, e2, e3)) show(e)
```

```
}
```

Обратите внимание на то, что приложение вполне работоспособно даже при отсутствии определения в нем метода `main`, поскольку оно является наследником трейта `App`. Запустить программу `Express` можно командой:

```
scala Express
```

При этом на выходном устройстве будет получен следующий вывод:

```
1
- * (x + 1)
2

x   1.5
- + ---
2    x
1
- * (x + 1)
2
-----------
x   1.5
- + ---
2    x
```

Резюме

В этой главе была представлена подробная информация о case-классах и поиске по шаблону. Их применение позволяет получить преимущества от использования ряда лаконичных идиом, которые обычно недоступны в объектно-ориентированных языках. Но реализованный в Scala поиск по шаблону не ограничивается тем,

что было показано в данной главе. Если нужно использовать поиск по шаблону, но при этом нежелательно открывать доступ к вашим классам, как это делается в case-классах, можно воспользоваться экстракторами, рассматриваемыми в главе 26. В следующей главе переключим свое внимание на списки.

93 Вместо абстрактного класса можно было бы выбрать моделирование корня этой иерархии классов в виде трейта. Использованное здесь моделирование в виде абстрактного класса может оказаться чуть-чуть эффективнее.

94 Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. — Addison-Wesley, 1995.

95 В типизированном шаблоне m: Map[_, _], часть "Map[_, _]" называется шаблоном типа.

96 Значение unaryPrecedence является самой высокой из возможных степенью приоритетности, поскольку ему было присвоено значение, на единицу большее значения степени приоритетности операторов * и %.

16. Работа со списками

Наиболее востребованными структурами данных в программах на Scala являются, наверное, списки. В этой главе подробно разъясняется, что такое списки. В ней будет представлено много общих операций, которые могут выполняться над списками. Также будут раскрыты некоторые наиболее важные принципы конструирования программ, работающих со списками.

16.1. Литералы списков

Списки уже попадались в предыдущих главах, следовательно, вам известно, что список, содержащий элементы `'a'`, `'b'` и `'c'`, записывается как `List('a', 'b', 'c')`. А вот другие примеры:

```
val fruit = List("apples", "oranges", "pears")
val nums = List(1, 2, 3, 4)
val diag3 =
  List(
    List(1, 0, 0),
    List(0, 1, 0),
    List(0, 0, 1)
  )
val empty = List()
```

Списки очень похожи на массивы, но у них имеются два важных отличия. Во-первых, списки являются неизменяемой структурой данных, то есть их элементы не могут быть изменены путем присваивания. Во-вторых, списки имеют рекурсивную структуру (имеется в виду *связанный список*)[97], а у массивов она линейная.

16.2. Тип списка

Как и массивы, списки однородны: у всех элементов списка один и тот же тип. Тип списка, имеющего элементы типа T, записывается как List[T]. Например, далее показаны те же четыре списка с добавленным явным указанием типов:

```
val fruit: List[String] = List("apples",
"oranges", "pears")
val nums: List[Int] = List(1, 2, 3, 4)
val diag3: List[List[Int]] =
  List(
    List(1, 0, 0),
    List(0, 1, 0),
    List(0, 0, 1)
  )
val empty: List[Nothing] = List()
```

В Scala тип списка обладает *ковариантностью*. Этот означает, что для каждой пары типов S и T, если S является подтипом T, то List[S] является подтипом List[T]. Например, List[String] является подтипом List[Object]. И это вполне естественно, поскольку каждый список строк также может рассматриваться как список объектов[98].

Заметьте, что типом пустого списка является List[Nothing]. В разделе 11.3 уже было показано, что Nothing — это низший тип в иерархии классов Scala. Он является подтипом любого другого имеющегося в Scala типа. Поскольку списки ковариантны, из этого следует, что List[Nothing] является подтипом List[T] для любого типа T, следовательно, пустой списочный объект, имеющий тип List[Nothing], также может рассматриваться в качестве объекта любого другого списочного типа, имеющего вид List[T]. Именно поэтому вполне допустимо создать следующий код:

```
// List() также относится к типу List[String]!
val xs: List[String] = List()
```

16.3. Создание списков

Все списки создаются из двух основных строительных блоков, `Nil` и `::` (произносится как «конс», от слова «конструировать»). `Nil` представляет пустой список. Инфиксный оператор конструирования `::` обозначает расширение списка по фронту. То есть запись `x :: xs` представляет собой список, чьим первым элементом является `x`, за которым следует элемент `xs` (или элементы списка `xs`). Следовательно, предыдущие списочные значения могут быть определены таким образом:

```
val fruit = "apples" :: ("oranges" :: ("pears" ::
Nil))
val nums  = 1 :: (2 :: (3 :: (4 :: Nil)))
val diag3 = (1 :: (0 :: (0 :: Nil))) ::
            (0 :: (1 :: (0 :: Nil))) ::
            (0 :: (0 :: (1 :: Nil))) :: Nil
val empty = Nil
```

Фактически предыдущие определения `fruit`, `nums`, `diag3` и `empty`, выраженные в виде `List(...)`, являются оболочками, расширяемыми в эти определения. Например, применение `List(1, 2, 3)` приводит к созданию списка `1 :: (2 :: (3 :: Nil))`.

То, что операция `::` заканчивается двоеточием, означает наличие у нее правой ассоциативности: `A :: B :: C` интерпретируется как `A :: (B :: C)`. Поэтому круглые скобки в предыдущих определениях можно отбросить. Например,

```
val nums = 1 :: 2 :: 3 :: 4 :: Nil
```

будет эквивалентом предыдущего определения `nums`.

16.4. Основные операции, проводимые над списками

Все действия со списками могут быть сведены к трем основным операциям:

- `head` возвращает первый элемент списка;
- `tail` возвращает список, состоящий из всех элементов, за исключением первого;
- `isEmpty` возвращает `true`, если список пуст.

Эти операции определены как методы класса `List`. Некоторые примеры их использования показаны в табл. 16.1. Методы `head` и `tail` определены только для непустых списков. Будучи примененными к пустому списку, они выдают исключение:

```
scala> Nil.head
java.util.NoSuchElementException: head of empty list
```

В качестве примера того, как можно обрабатывать список, рассмотрим сортировку элементов списка чисел в возрастающем порядке. Один из простых способов выполнения этой задачи заключается во *вставочной сортировке*, которая работает следующим образом: для сортировки непустого списка `x :: xs` сортируется остаток `xs` и первый элемент `x` вставляется в нужную позицию результата.

Таблица 16.1. Основные операции со списками

Что используется	Что этот метод делает
empty.isEmpty	Возвращает true
fruit.isEmpty	Возвращает false
fruit.head	Возвращает "apples"
fruit.tail.head	Возвращает "oranges"

diag3.head	Возвращает List(1, 0, 0)

Сортировка пустого списка выдает пустой список. Выраженный в виде кода Scala, алгоритм вставочной сортировки выглядит следующим образом:

```
def isort(xs: List[Int]): List[Int] =
  if (xs.isEmpty) Nil
  else insert(xs.head, isort(xs.tail))

def insert(x: Int, xs: List[Int]): List[Int] =
  if (xs.isEmpty || x <= xs.head) x :: xs
  else xs.head :: insert(x, xs.tail)
```

16.5. Шаблоны-списки

Разбор списков можно выполнять также с помощью поиска по шаблону. Шаблоны-списки по порядку следования соответствуют выражениям списков. С помощью шаблона вида `List(...)` можно либо выявлять соответствие всем элементам списка, либо рассматривать список разрозненно, поэлементно, используя шаблоны, составленные из оператора `::` и константы `Nil`.

Пример первой разновидности шаблона выглядит следующим образом:

```
scala> val List(a, b, c) = fruit
a: String = apples
b: String = oranges
c: String = pears
```

Шаблон `List(a, b, c)` соответствует спискам длиной в три элемента и привязывает эти три элемента к шаблонам-переменным `a`, `b` и `c`. Если количество элементов заранее неизвестно, то лучше вместо этого искать соответствие с

использованием оператора `::`. Например, шаблон `a :: b :: rest` соответствует спискам длиной два и более элементов:

```
scala> val a
:: b
:: rest = fruit
a: String = apples
b: String = oranges
rest: List[String] = List(pears)
```

О поиске по шаблону в объектах класса LIST

Если провести беглый обзор возможных форм шаблонов, рассмотренных в главе 15, выяснится, что ничего похожего ни на `List(...)`, ни на `::` в определенных там разновидностях нет. Фактически `List(...)` является экземпляром определенного в библиотеке шаблона-экстрактора. Такие шаблоны будут рассматриваться в главе 26. Конс-шаблон `x :: xs` является особым случаем шаблона инфиксной операции. В качестве выражения инфиксная операция является эквивалентом вызова метода. Для шаблонов действуют иные правила: в качестве шаблона такая инфиксная операция, как `p op q`, является эквивалентом `op(p, q)`. То есть инфиксный оператор `op` рассматривается в качестве шаблона-конструктора. В частности, такой конс-шаблон, как `x :: xs`, рассматривается как `::(x, xs)`.

Это обстоятельство подсказывает нам, что должен быть класс по имени `::`, который соответствует шаблону-конструктору. Разумеется, подобный класс существует — он называется `scala.::`, и это именно тот класс, который создает непустые

списки. Следовательно, `::` в Scala фигурирует дважды: как имя класса в пакете `scala` и как метод в классе `List`. Задачей метода `::` является создание экземпляра класса `scala.::`. Более подробно реализация класса `List` рассматривается в главе 22.

Извлечение части списков с помощью шаблонов является альтернативой использованию основных методов `head`, `tail` и `isEmpty`. Например, в данном коде снова применена вставочная сортировка, на этот раз записанная с поиском по шаблону:

```
def isort(xs: List[Int]): List[Int] = xs match {
  case List()   => List()
  case x :: xs1 => insert(x, isort(xs1))
}

def insert(x: Int, xs: List[Int]): List[Int] = xs
match {
  case List()  => List(x)
  case y :: ys => if (x <= y) x :: xs
                  else y :: insert(x, ys)
}
```

Зачастую применение к спискам поиска по шаблону оказывается понятнее, чем их декомпозиция с помощью методов, поэтому поиск по шаблону должен стать частью вашего инструментария для обработки списков.

Вот и все, что нужно знать о списках в Scala, чтобы их правильно применять. Но существует также множество методов, которые вбирают в себя наиболее распространенные схемы проведения операций со списками. Эти методы делают программы обработки списков лаконичнее и зачастую понятнее. В следующих

двух разделах будут представлены наиболее важные методы, определенные в классе List.

16.6. Методы первого порядка, определенные в классе List

В этом разделе рассматривается большинство методов *первого порядка*, определенных в классе List. Метод относится к первому порядку, если он не получает в качестве аргументов никаких функций. Здесь с использованием двух примеров также будет представлен ряд рекомендованных приемов структурирования программ, работающих со списками.

Объединение двух списков

Операцией, похожей на `::`, является объединение списков, записываемое в виде `:::`. В отличие от операции `::`, операция `:::` получает в качестве операндов два списка. Результатом выполнения кода `xs ::: ys` является новый список, содержащий все элементы списка `xs`, за которыми следуют все элементы списка `ys`.

Рассмотрим несколько примеров:

```
scala> List(1, 2) ::: List(3, 4, 5)
res0: List[Int] = List(1, 2, 3, 4, 5)

scala> List() ::: List(1, 2, 3)
res1: List[Int] = List(1, 2, 3)

scala> List(1, 2, 3) ::: List(4)
res2: List[Int] = List(1, 2, 3, 4)
```

Как и конс-оператор, объединение списков обладает правой ассоциативностью. Такое выражение, как:

```
xs ::: ys ::: zs
```

интерпретируется следующим образом:

```
xs ::: (ys ::: zs)
```

Принцип «разделяй и властвуй»

Объединение (`:::`) реализовано в качестве метода в классе `List`. Можно было бы также реализовать объединение собственными силами, используя в отношении списков поиск по шаблону. Попытка самостоятельного решения этой задачи будет весьма поучительной, поскольку в ходе ее выполнения мы покажем основные способы реализации алгоритмов с использованием списков. Во-первых, будет освоена сигнатура для метода объединения, который мы назовем `append`. Чтобы не создавать большой путаницы, предположим, что `append` определен за пределами класса `List`, поэтому он будет получать в качестве параметров два объединяемых списка. Эти два списка должны быть согласованы по типу их элементов, но сам этот тип может быть произвольным. Все это может быть обеспечено путем задания `append` параметра типа[99], представляющего тип элементов двух входящих списков:

```
def append[T](xs: List[T], ys: List[T]): List[T]
```

Чтобы сконструировать реализацию `append`, есть смысл вспомнить принцип конструирования «разделяй и властвуй» для программ, работающих с такими рекурсивными структурами данных, как списки. Многие алгоритмы для работы со списками сначала разбивают входящий список на простые блоки, используя для этого поиск по шаблону. Это часть принципа, которая называется *разделяй*. Затем выстраивается результат для каждого блока. Если результатом является непустой список, некоторые из его частей могут быть выстроены путем рекурсивных вызовов того

же самого алгоритма. В этом будет заключаться та часть принципа, которая называется *властвуй*.

Чтобы применить этот принцип к реализации метода `append`, сначала стоит ответить на вопрос: какому именно списку нужно соответствовать? Применительно к `append` данный вопрос не настолько прост, как для для многих других методов, поскольку здесь существуют два варианта. Но следующая далее фаза «властвования» подсказывает, что нужно выстроить список, состоящий из всех элементов обоих входящих списков. Поскольку списки выстраиваются из конца в начало, `ys` можно оставить нетронутым, а вот `xs` нужно разбирать и пристраивать впереди `ys`. Таким образом, есть смысл сконцентрироваться на `xs` как на источнике поиска по шаблону. Чаще всего поиск по шаблону в отношении списков выполняется, чтобы отличить пустой список от непустого. Таким образом, для метода `append` вырисовывается следующая схема:

```
def append[T](xs: List[T], ys: List[T]): List[T] =
  xs match {
    case List() => ???
    case x :: xs1 => ???
  }
```

Остается лишь заполнить два места с пометками `???`[100]. Первым таким местом представлен выбор для варианта, когда входящий список `xs` пустой. В таком случае в результате объединения получается второй список:

```
case List() => ys
```

Вторым местом, оставленным незаполненным, представлен выбор для случая, когда входящий список `xs` состоит из некоего заголовка `x`, за которым следует остальная часть `xs1`. В таком случае результатом также будет непустой список. Чтобы выстроить непустой список, нужно знать, какой должна быть его «голова»

(head), а каким — «хвост» (tail). Вам известно, что первым элементом получающегося в результате списка является `x`. Что касается остальных элементов, они могут быть вычислены путем добавления второго списка `ys` к оставшейся части первого списка `xs1`.

Это позволяет завершить конструкцию и получить следующий код:

```
def append[T](xs: List[T], ys: List[T]): List[T] =
  xs match {
    case List() => ys
    case x :: xs1 => x :: append(xs1, ys)
  }
```

Вычисление второго варианта является иллюстрацией части «властвуй» принципа «разделяй и властвуй»: сначала нужно продумать форму желаемых результатов, затем вычислить отдельные части этой формы, используя где только можно рекурсивные вызовы алгоритма. И наконец, составить из этих частей вывод.

Получение длины списка: length

Метод `length` вычисляет длину списка:

```
scala> List(1, 2, 3).length
res3: Int = 3
```

Определение длины списков, в отличие от массивов, является довольно затратной операцией. Для ее выполнения требуется пройти по всему списку в поисках его конца, и на это затрачивается время, пропорциональное количеству элементов списка. Поэтому вряд ли имеет смысл подменять `xs.isEmpty` выражением `xs.length == 0`. Результаты окажутся одинаковыми, но второе выражение будет выполняться медленнее, в частности

если список xs имеет большую длину.

Обращение к концу списка: init и last

Теперь вам уже известны основные операции head и tail, в результате выполнения которых извлекаются, соответственно, первый элемент списка и весь остальной список, за исключением первого элемента. У каждой из них имеется обратная по смыслу операция: last возвращает последний элемент непустого списка, а init — список, состоящий из всех элементов, за исключением последнего:

```
scala> val abcde = List('a', 'b', 'c', 'd', 'e')
abcde: List[Char] = List(a, b, c, d, e)

scala> abcde.last
res4: Char = e

scala> abcde.init
res5: List[Char] = List(a, b, c, d)
```

Подобно методам head и tail, эти методы, примененные к пустому списку, выдают исключение:

```
scala> List().init
java.lang.UnsupportedOperationException: Nil.init
      at scala.List.init(List.scala:544)
      at ...

scala> List().last
java.util.NoSuchElementException: Nil.last
      at scala.List.last(List.scala:563)
      at ...
```

В отличие от `head` и `tail`, на выполнение которых неизменно затрачивается одно и то же время, методам `init` и `last` для вычисления результата необходимо обойти весь список. То есть для их выполнения нужно время, пропорциональное длине списка.

СОВЕТ

Неплохо было бы организовать ваши данные таким образом, чтобы основная часть обращений приходилась на начальный, а не на последний элемент списка.

Реверсирование списков: reverse

Если в какой-то момент вычисления алгоритма требуются быстрые обращения к концу списка, порой будет разумнее сначала перестроить список в обратном порядке, а потом уже работать с его результатами. Получить такой список можно следующим образом:

```
scala> abcde.reverse
res6: List[Char] = List(e, d, c, b, a)
```

Как и все остальные операции со списками, `reverse` создает новый список, а не изменяет тот, с которым работает. Поскольку списки неизменяемы, сделать это все равно бы было невозможно. Чтобы убедиться в этом, проверьте, что исходный список `abcde` после операции `reverse` не изменился:

```
scala> abcde
res7: List[Char] = List(a, b, c, d, e)
```

Операции `reverse`, `init` и `last` подчиняются ряду законов, которыми можно воспользоваться для продумывания вычислений и упрощения программ.

- `reverse` производит собственную противоположность:

```
xs.reverse.reverse является эквивалентом xs
```

- `reverse` превращает `init` в `tail`, а `last` в `head`, за исключением того, что все элементы стоят в обратном порядке:

```
xs.reverse.init       является       эквивалентом
xs.tail.reverse
xs.reverse.tail       является       эквивалентом
xs.init.reverse
xs.reverse.head является эквивалентом xs.last
xs.reverse.last является эквивалентом xs.head
```

Реверсирование может быть реализовано с использованием объединения (`:::`), как в следующем методе по имени `rev`:

```
def rev[T](xs: List[T]): List[T] = xs match {
  case List() => xs
  case x :: xs1 => rev(xs1) ::: List(x)
}
```

Но этот метод, вопреки предположениям, менее эффективен. Чтобы убедиться в высокой вычислительной сложности `rev`, представьте, что список `xs` имеет длину `n`. Учтите, что придется делать `n` рекурсивных вызовов `rev`. Каждый вызов, за исключением последнего, влечет за собой объединение списков. На объединение `xs ::: ys` затрачивается время, пропорциональное длине его первого аргумента `xs`. Следовательно, общая вычислительная сложность `rev` выражается следующим образом:

```
n + (n - 1) + ... + 1 = (1 + n) * n / 2
```

Иными словами, вычислительная сложность `rev` выражается уравнением второй степени от длины его входящего аргумента. Это обстоятельство, когда сравниваешь его со стандартным

реверсированием изменяемого связанного списка, имеющего линейную вычислительную сложность, вызывает разочарование. Но данная реализация `rev` не является наилучшей из возможных. На примере, который начинается здесь, вы увидите, как все это можно ускорить.

Префиксы и суффиксы: drop, take и splitAt

Операции `drop` и `take` обобщают `tail` и `init` в том смысле, что возвращают произвольные префиксы или суффиксы списка. Выражение `xs take n` возвращает первые `n` элементов списка `xs`. Если `n` больше `xs.length`, возвращается весь список `xs`. Операция `xs drop n` возвращает все элементы списка `xs`, за исключением первых `n` элементов. Если `n` больше `xs.length`, возвращается пустой список.

Операция `splitAt` разбивает список по заданному индексу, возвращая пару из двух списков[101]. Она определяется в виде следующего эквивалента:

```
xs splitAt n является эквивалентом (xs take n, xs
drop n)
```

Но операция `splitAt` избегает двойного прохода элементов списка. Примеры применения этих трех методов выглядят следующим образом:

```
scala> abcde take 2
res8: List[Char] = List(a, b)

scala> abcde drop 2
res9: List[Char] = List(c, d, e)

scala> abcde splitAt 2
res10:   (List[Char],   List[Char])   =   (List(a,
```

```
b),List(c, d, e))
```

Выбор элемента: apply и indices

Произвольный выбор элемента поддерживается методом `apply`, но эта операция менее востребована, чем подобная операция для массивов:

```
scala> abcde apply 2 // в Scala используется довольно редко
res11: Char = c
```

Что же касается всех остальных типов, то подразумевается, что `apply` вставляется в вызове метода, когда объект появляется в позиции функции. Поэтому показанную ранее строку кода можно сократить до следующей:

```
scala> abcde(2) // в Scala используется довольно редко
res12: Char = c
```

Одной из причин того, что выбор произвольного элемента менее популярен для списков, чем для массивов, является то, что на выполнение кода `xs(n)` затрачивается время, пропорциональное величине значения индекса `n`. Фактически метод `apply` определен сочетанием методов `drop` и `head`:

```
xs apply n является эквивалентом (xs drop n).head
```

Из этого определения также становится понятно, что индексы списков, как и индексы массивов, задаются в диапазоне от 0 до длины списка минус один. Метод `indices` возвращает список, состоящий из всех допустимых индексов заданного списка:

```
scala> abcde.indices
res13: scala.collection.immutable.Range
```

```
  = Range(0, 1, 2, 3, 4)
```

Линеаризация списка списков: flatten

Метод `flatten` получает список списков и линеаризирует его в единый список:

```
scala> List(List(1, 2), List(3), List(), List(4, 5)).flatten
res14: List[Int] = List(1, 2, 3, 4, 5)

scala> fruit.map(_.toCharArray).flatten
res15: List[Char] = List(a, p, p, l, e, s, o, r, a, n, g, e,
s, p, e, a, r, s)
```

Он применяется только к тем спискам, все элементы которых являются списками. Попытка линеаризации других списков приведет к выдаче ошибки компиляции:

```
scala> List(1, 2, 3).flatten
<console>:8: error: No implicit view available from Int =>
scala.collection.GenTraversableOnce[B].
              List(1, 2, 3).flatten
                            ^
```

Объединение в пары и обратное разбиение: zip и unzip

Операция `zip` получает два списка и формирует список из пар их значений:

```
scala> abcde.indices zip abcde
res17: scala.collection.immutable.IndexedSeq[(Int, Char)] =
```

```
Vector((0,a), (1,b), (2,c), (3,d), (4,e))
```

Если списки разной длины, все элементы без пары отбрасываются:

```
scala> val zipped = abcde zip List(1, 2, 3)
zipped: List[(Char, Int)] = List((a,1), (b,2), (c,3))
```

Особо полезным вариантом является объединение в пары списка с его индексами. Наиболее эффективно это выполняется с помощью метода `zipWithIndex`, составляющего пары из каждого элемента списка и той позиции, в которой он появляется в этом списке.

```
scala> abcde.zipWithIndex
res18: List[(Char, Int)] = List((a,0), (b,1), (c,2), (d,3), (e,4))
```

Любой список кортежей может быть превращен обратно в кортеж списков с помощью метода `unzip`:

```
scala> zipped.unzip
res19: (List[Char], List[Int])
  = (List(a, b, c),List(1, 2, 3))
```

Методы `zip` и `unzip` реализуют один из способов одновременной работы с несколькими списками. Более лаконичный способ задания такой же работы показан в разделе 16.9.

Отображение списков: toString и mkString

Операция `toString` возвращает каноническое строковое представление списка:

```
scala> abcde.toString
res20: String = List(a, b, c, d, e)
```

Если требуется иное представление, можно воспользоваться методом mkString. Операция xs mkString (pre, sep, post) задействует четыре операнда: отображаемый список xs, префиксную строку pre, отображаемую перед всеми элементами, строковый разделитель sep, отображаемый между последовательно выводимыми элементами, и постфиксную строку, отображаемую в конце.

Результатом операции будет следующая строка:

```
pre + xs(0) + sep + ...+ sep + xs(xs.length - 1) +
post
```

У метода mkString имеются два перегружаемых варианта, которые позволяют отбрасывать некоторые или все его аргументы. Первый вариант получает только строковый разделитель:

```
xs mkString sep является эквивалентом xs mkString
("", sep, "")
```

Второй вариант позволяет опустить все аргументы:

```
xs.mkString является эквивалентом xs mkString ""
```

Рассмотрим несколько примеров:

```
scala> abcde mkString ("[", ",", "]")
res21: String = [a,b,c,d,e]

scala> abcde mkString ""
res22: String = abcde

scala> abcde.mkString
res23: String = abcde
```

```
scala> abcde mkString ("List(", ", ", ")")
res24: String = List(a, b, c, d, e)
```

Есть также варианты методов `mkString`, называющиеся `addString`, которые не возвращают созданную строку в качестве результата, а добавляют ее к объекту `StringBuilder`[102]:

```
scala> val buf = new StringBuilder
buf: StringBuilder =

scala> abcde addString (buf, "(", ";", ")")
res25: StringBuilder = (a;b;c;d;e)
```

Методы `mkString` и `addString` наследуются из родительского трейта `Traversable` класса `List`, поэтому их можно применять ко всем другим коллекциям.

Преобразование списков: iterator, toArray, copyToArray

Чтобы выполнить преобразование данных между линейным миром массивов и рекурсивным миром списков, можно воспользоваться методом `toArray` в классе `List` и методом `toList` в классе `Array`:

```
scala> val arr = abcde.toArray
arr: Array[Char] = Array(a, b, c, d, e)

scala> arr.toList
res26: List[Char] = List(a, b, c, d, e)
```

Есть также метод `copyToArray`, который копирует элементы списка в последовательные позиции массива внутри некоего массива назначения. Операция

```
xs copyToArray (arr, start)
```

копирует все элементы списка `xs` в массив `arr`, начиная с позиции `start`. Нужно обеспечить достаточную длину массива назначения `arr`, чтобы в нем мог поместиться весь список. Рассмотрим пример:

```
scala> val arr2 = new Array[Int](10)
arr2: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

scala> List(1, 2, 3) copyToArray (arr2, 3)

scala> arr2
res28: Array[Int] = Array(0, 0, 0, 1, 2, 3, 0, 0, 0, 0)
```

И наконец, если нужно получить доступ к элементам списка через итератор, можно воспользоваться методом `iterator`:

```
scala> val it = abcde.iterator
it: Iterator[Char] = non-empty iterator

scala> it.next
res29: Char = a

scala> it.next
res30: Char = b
```

Пример: сортировка методом слияния

Ранее представленная сортировка методом вставки записывается кратко, но эффективность ее невысока. Ее усредненная вычислительная сложность пропорциональна квадрату длины входящего списка. А более эффективным алгоритмом обладает сортировка *методом слияния*.

ускоренный режим чтения

Этот пример — еще одна иллюстрация карринга и принципа разделения и властвования. Также в нем говорится о вычислительной сложности алгоритма, что может оказаться полезным. Если же читать книгу нужно быстрее, раздел 16.7 можно спокойно пропустить.

Сортировка методом слияния работает следующим образом: если список имеет один элемент или не имеет никаких элементов, то он уже отсортирован, поэтому может быть возвращен в исходном виде. Более длинные списки разбиваются на два подсписка, в каждом из которых содержится около половины элементов исходного списка. Каждый подсписок сортируется путем рекурсивного вызова функции `sort`, затем два отсортированных списка объединяются в ходе операции слияния.

Для обобщенной реализации сортировки методом слияния вам придется оставить открытыми тип элементов сортируемого списка и функцию, которая будет использоваться для сравнения элементов. Максимально обобщенную функцию можно получить, передав ей эти два элемента в качестве параметров. При этом получится реализация, показанная в листинге 16.1.

Листинг 16.1. Функция сортировки методом слияния для List-объектов

```
def msort[T](less: (T, T) => Boolean)
    (xs: List[T]): List[T] = {
  def merge(xs: List[T], ys: List[T]): List[T] =
    (xs, ys) match {
      case (Nil, _) => ys
      case (_, Nil) => xs
      case (x :: xs1, y :: ys1) =>
```

```
        if (less(x, y)) x :: merge(xs1, ys)
        else y :: merge(xs, ys1)
      }

    val n = xs.length / 2
    if (n == 0) xs
    else {
      val (ys, zs) = xs splitAt n
      merge(msort(less)(ys), msort(less)(zs))
    }
  }
```

Вычислительная сложность `msort` задается как $n \log(n)$, где n — длина входящего списка. Чтобы понять причину происходящего, следует отметить, что и разбиение списка на два подсписка, и слияние двух отсортированных списков требуют времени, которое пропорционально длине аргумента `list(s)`. Каждый рекурсивный вызов `msort` вполовину уменьшает количество элементов, используемых им в качестве входящих данных, поэтому производится примерно $\log(n)$ последовательных вызовов, выполняемых до тех пор, пока не будет достигнут базовый вариант списков длиной в единицу. Но для более длинных списков каждый вызов порождает два последующих вызова. Если все это сложить вместе, получится, что при любом уровне вызова $\log(n)$ каждый элемент исходных списков примет участие в одной операции разбиения и одной операции слияния.

Следовательно, каждый уровень вызова имеет общий уровень затрат, пропорциональный n. Поскольку число уровней вызова равно $\log(n)$, мы получаем общий уровень затрат, пропорциональный $n \log(n)$. Этот уровень затрат не зависит от исходного распределения элементов в списке, следовательно, в наихудшем варианте он будет таким же, как уровень затрат в усредненном варианте. Это свойство делает сортировку методом

слияния весьма привлекательным алгоритмом для сортировки списков.

Пример использования `msort` выглядит следующим образом:

```
scala> msort((x: Int, y: Int) => x < y)(List(5, 7,
1, 3))
res31: List[Int] = List(1, 3, 5, 7)
```

Функция `msort` представляет собой классический пример концепции карринга, рассмотренной в разделе 9.3. Каррирование упрощает специализацию функции для конкретных функций сравнения. Рассмотрим пример:

```
scala> val intSort = msort((x: Int, y: Int) => x <
y) _
intSort: List[Int] => List[Int] = <function1>
```

Переменная `intSort` ссылается на функцию, получающую список целочисленных значений и сортирующую их в порядке следования чисел. Как объяснялось в разделе 8.6, знак подчеркивания означает недостающий список аргументов. В данном случае в качестве недостающего аргумента фигурирует сортируемый список. А вот другой пример, демонстрирующий способ возможного определения функции, выполняющей сортировку списка целочисленных значений в обратном порядке следования чисел:

```
scala> val reverseIntSort = msort((x: Int, y: Int)
=> x > y) _
reverseIntSort: (List[Int]) => List[Int] =
<function>
```

Поскольку функция сравнения уже представлена посредством карринга, при вызове функций `intSort` или `reverseIntSort` нужно будет только предоставить сортируемый список.

Рассмотрим несколько примеров:

```
scala> val mixedInts = List(4, 1, 9, 0, 5, 8, 3,
6, 2, 7)
mixedInts: List[Int] = List(4, 1, 9, 0, 5, 8, 3,
6, 2, 7)

scala> intSort(mixedInts)
res0: List[Int] = List(0, 1, 2, 3, 4, 5, 6, 7, 8,
9)

scala> reverseIntSort(mixedInts)
res1: List[Int] = List(9, 8, 7, 6, 5, 4, 3, 2, 1,
0)
```

16.7. Методы высшего порядка, определенные в классе List

У многих операций над списками схожая структура. Раз за разом используются несколько схем. К подобным примерам можно отнести какое-либо преобразование каждого элемента списка, проверку того, что свойство соблюдается для всех элементов списка, извлечение из списка элементов, удовлетворяющих каким-то критериям, или объединение элементов списка с использованием какого-нибудь оператора. В Java подобные схемы будут, как правило, созданы идиоматическими комбинациями циклов `for` или `while`. В Scala они могут быть выражены короче и непосредственнее за счет использования операторов высшего порядка[103], которые реализуются в виде методов, определенных в классе `List`. Этим операторам высшего порядка и посвящен данный раздел.

Отображение, применяемое в отношении списка: map, flatMap и foreach

Операция `xs map f` получает в качестве операндов список `xs` типа

List[T] и функцию f типа T => U. Она возвращает список, получающийся в результате применения f к каждому элементу списка xs, например:

```
scala> List(1, 2, 3) map (_ + 1)
res32: List[Int] = List(2, 3, 4)

scala> val words = List("the", "quick", "brown",
"fox")
words: List[String] = List(the, quick, brown, fox)

scala> words map (_.length)
res33: List[Int] = List(3, 5, 5, 3)

scala> words map (_.toList.reverse.mkString)
res34: List[String] = List(eht, kciuq, nworb, xof)
```

Оператор flatMap похож на map, но в качестве правого операнда получает функцию, возвращающую список элементов. Он применяет функцию к каждому элементу списка и возвращает объединение всех результатов выполнения функции. Разница между map и flatMap показана в следующем примере:

```
scala> words map (_.toList)
res35: List[List[Char]] = List(List(t, h, e),
List(q, u, i,
    c, k), List(b, r, o, w, n), List(f, o, x))

scala> words flatMap (_.toList)
res36: List[Char] = List(t, h, e, q, u, i, c, k,
b, r, o, w,
    n, f, o, x)
```

Как видите, там, где map возвращает список списков, flatMap

возвращает единый список, в котором объединены все элементы списков.

Различия и взаимодействие методов `map` и `flatMap` показаны также следующим выражением, с помощью которого создается список всех пар `(i, j)`, отвечающих условию $1 \leqslant j < i < 5$:

```
scala> List.range(1, 5) flatMap (
         i => List.range(1, i) map (j => (i, j))
       )
res37: List[(Int, Int)] = List((2,1), (3,1),
(3,2), (4,1),
   (4,2), (4,3))
```

`List.range` является вспомогательным методом, создающим список из всех целых чисел в некотором диапазоне. В этом примере он используется дважды: первый раз — для создания списка целых чисел от 1 (включительно) до 5 (исключительно), второй раз — для создания списка целых чисел от 1 до `i` для каждого значения `i`, взятого из первого списка. Метод `map` в этом выражении создает список кортежей `(i, j)`, где `j < i`. Охватывающий метод `flatMap` в примере создает такой список для каждого `i` между 1 и 5, а затем объединяет все результаты. По-другому этот же список может быть создан с использованием выражения `for`:

```
for (i <- List.range(1, 5); j <- List.range(1, i))
yield (i, j)
```

Взаимодействие выражений `for` и операций со списками более подробно будет рассмотрено в главе 23.

Третьей `map`-подобной операцией является `foreach`. Но, в отличие от `map` и `flatMap`, операция `foreach` получает в качестве правого операнда процедуру (функцию, результатом которой является тип `Unit`). Она просто применяет процедуру к каждому

элементу списка. А сам результат операции также имеет тип Unit, то есть никакого сбора списка результатов не происходит. В качестве примера рассмотрим краткий способ суммирования всех чисел, имеющихся в списке:

```
scala> var sum = 0
sum: Int = 0

scala> List(1, 2, 3, 4, 5) foreach (sum += _)

scala> sum
res39: Int = 15
```

Фильтрация списков: filter, partition, find, takeWhile, dropWhile и span

Операция `xs filter p` получает в качестве операндов список `xs` типа `List[T]` и функцию-предикат `p`, относящуюся к типу `T => Boolean`. Эта операция выдает список всех элементов `x` из списка `xs`, для которых `p(x)` вычисляется в `true`, например:

```
scala> List(1, 2, 3, 4, 5) filter (_ % 2 == 0)
res40: List[Int] = List(2, 4)

scala> words filter (_.length == 3)
res41: List[String] = List(the, fox)
```

Метод `partition` похож на метод `filter`, но возвращает пару списков. Один список содержит все элементы, для которых предикат вычисляется в `true`, а другой — все элементы, для которых предикат вычисляется в `false`. Его можно определить путем использования эквивалента:

```
xs partition p является эквивалентом (xs filter p,
xs filter (!p(_)))
```

Пример его работы выглядит следующим образом:

```
scala> List(1, 2, 3, 4, 5) partition (_ % 2 == 0)
res42: (List[Int], List[Int]) = (List(2, 4),List(1, 3, 5))
```

Метод `find` также похож на метод `filter`, но возвращает только первый элемент, удовлетворяющий условию заданного предиката, а не все такие элементы. Операция `xs find p` получает в качестве операндов список `xs` и предикат `p`. Она возвращает значение, которое может отсутствовать. Если в списке `xs` есть элемент `x`, для которого `p(x)` вычисляется в `true`, то возвращается `Some(x)`. В противном случае `p` вычисляется в `false` для всех элементов и возвращается `None`. Вот несколько примеров работы этого метода:

```
scala> List(1, 2, 3, 4, 5) find (_ % 2 == 0)
res43: Option[Int] = Some(2)

scala> List(1, 2, 3, 4, 5) find (_ <= 0)
res44: Option[Int] = None
```

Операторы `takeWhile` и `dropWhile` также получают в качестве правого операнда предикат. Операция `xs takeWhile p` получает самый длинный префикс списка `xs`, в котором каждый элемент удовлетворяет условию предиката `p`. Аналогично этому операция `xs dropWhile p` удаляет самый длинный префикс из списка `xs`, в котором каждый элемент удовлетворяет условию предиката `p`. Ряд примеров использования этого метода выглядит следующим образом:

```
scala> List(1, 2, 3, -4, 5) takeWhile (_ > 0)
res45: List[Int] = List(1, 2, 3)
```

```
scala> words dropWhile (_ startsWith "t")
res46: List[String] = List(quick, brown, fox)
```

Метод span объединяет takeWhile и dropWhile в одну операцию точно так же, как метод splitAt объединяет stake и drop. Он возвращает пару из двух списков, определяемых следующим эквивалентом:

```
xs span p является эквивалентом (xs takeWhile p,
xs dropWhile p)
```

Как и splitAt, метод span избегает двойного прохода элементов списка:

```
scala> List(1, 2, 3, -4, 5) span (_ > 0)
res47: (List[Int], List[Int]) = (List(1, 2,
3),List(-4, 5))
```

Применение предикатов к спискам: forall и exists

Операция xs forall p получает в качестве аргументов список xs и предикат p. Она возвращает результат true, если все элементы списка удовлетворяют условию предиката p. И наоборот, операция xs exists p возвращает true, если в xs имеется элемент, удовлетворяющий условию предиката p. Например, чтобы определить, есть ли в матрице, представленной списком списков, строка, состоящая только из нулевых элементов, можно применить следующий код:

```
scala> def hasZeroRow(m: List[List[Int]]) =
         m exists (row => row forall (_ == 0))
hasZeroRow: (m: List[List[Int]])Boolean

scala> hasZeroRow(diag3)
res48: Boolean = false
```

Свертка списков: /: и :\

Еще один распространенный вид операции сочетает в себе элементы списка с оператором, например:

```
sum(List(a, b, c)) является эквивалентом 0 + a + b
+ c
```

Это особый случай операции свертки:

```
scala> def sum(xs: List[Int]): Int = (0 /: xs) (_
+ _)
sum: (xs: List[Int])Int
```

Аналогично этому

```
product(List(a, b, c)) является эквивалентом 1 * a
* b * c
```

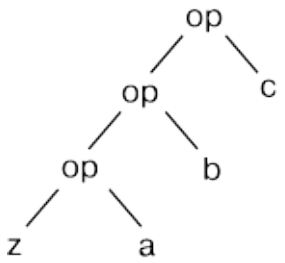
представляет собой особый случай операции свертки:

```
scala> def product(xs: List[Int]): Int = (1 /: xs)
(_ * _)
product: (xs: List[Int])Int
```

Операция левой свертки `(z /: xs) (op)` задействует три объекта: начальное значение `z`, список `xs` и бинарную операцию `op`. Результатом свертки является применение `op` между последовательно извлекаемыми элементами списка, где в качестве префикса выступает значение `z`, например:

```
(z /: List(a, b, c)) (op) является эквивалентом
op(op(op(z, a), b), c)
```

Или в графическом представлении:



Вот еще один пример, иллюстрирующий использование оператора `/:`. Для объединения всех слов в списке из строковых значений с пробелами между ними и пробелом в самом начале списка можно воспользоваться следующим кодом:

```
scala> ("" /: words) (_ + " " + _)
res49: String = " the quick brown fox"
```

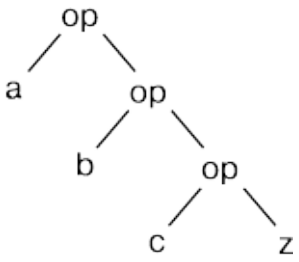
Этот код выдаст лишний пробел в самом начале. Избавиться от него поможет слегка видоизмененный вариант кода:

```
scala> (words.head /: words.tail) (_ + " " + _)
res50: String = the quick brown fox
```

Оператор `/:` создает деревья операций с уклоном влево (это отображается его синтаксисом с как бы поднимающимся вперед слешем). По аналогии с этим оператор `:\` создает деревья операций с уклоном вправо, например:

```
(List(a, b, c) :\ z) (op) является эквивалентом
op(a, op(b, op(c, z)))
```

Или в графическом представлении:



Оператор :\ (произносится «правая свертка») задействует те же три операнда, что и оператор левой свертки, но два из них указываются в обратном порядке: первым операндом является свертываемый список, а вторым — начальное значение.

Для ассоциативных операций левая и правая свертки абсолютно эквивалентны, но могут существовать различия в эффективности их применения. Рассмотрим, к примеру, операцию, относящуюся к методу `flatten`, которая объединяет все элементы в списке списков. Она может быть реализована с использованием либо левой, либо правой свертки:

```
def flattenLeft[T](xss: List[List[T]]) =
    (List[T]() /: xss) (_ ::: _)

def flattenRight[T](xss: List[List[T]]) =
    (xss :~List[T]()) (_ ::: _)
```

Поскольку объединение списков `xs ::: ys` занимает время, пропорциональное длине его первого аргумента `xs`, реализация в понятиях правой свертки в `flattenRight` более эффективна, чем реализация с применением левой свертки в `flattenLeft`. Дело в том, что `flattenLeft(xss)` копирует первый элемент списка `xss.head` n – 1 раз, где n — длина списка `xss`.

Учтите, что обе версии `flatten` требуют аннотации типа в отношении пустого списка, являющегося начальным значением свертки. Этот связано с ограничениями в имеющемся в Scala механизме вывода типа, которой не в состоянии автоматически вывести правильный тип списка. При попытке игнорировать аннотацию будет выдано следующее сообщение об ошибке:

```
scala> def flattenRight[T](xss: List[List[T]]) =
           (xss :~List()) (_ ::: _)
<console>:8: error: type mismatch;
found   : List[T]
```

```
required: List[Nothing]
           (xss :~List()) (_ ::: _)
                                  ^
```

Чтобы понять, почему не был выполнен надлежащий вывод типа, нужно узнать о типах методов свертки и способах их реализации. Более подробно этот вопрос рассматривается в разделе 16.10. И наконец, хотя операторы `/:` и `:\` обладают тем преимуществом, что направлениями слешей напоминают графическое изображение своих деревьев операций, наклоненных вправо или влево соответственно, свойства ассоциативности знака двоеточия приводят к тому, что начальное значение помещается в выражении в ту же позицию, что и в дереве, что некоторым на интуитивном уровне может показаться непонятным. При желании вместо них можно воспользоваться методами с именами `foldLeft` и `foldRight`, которые также определены в классе `List`.

Пример: реверсирование списков с помощью свертки

Ранее в этой главе была показана реализация метода реверсирования по имени `rev`, время работы которой вычислялось с помощью уравнения второй степени от длины реверсируемого списка. Теперь будет показана другая реализация метода реверсирования, затраты времени на выполнение которой имеют линейную зависимость. Идея состоит в том, чтобы воспользоваться операцией левой свертки, основанной на следующей схеме:

```
def reverseLeft[T](xs: List[T]) =
(стартовое_значение /: xs)(операция)
```

Остается заполнить части стартовое_значение и операция. Собственно, вы можете попытаться вывести эти части из нескольких простых примеров. Чтобы правильно отобразить стартовое_значение, можно начать с наименьшего возможного списка, `List()` и рассуждать следующим образом:

```
List()
```

является эквивалентом (в понятиях свойств `reverseLeft`):

```
reverseLeft(List())
```

является эквивалентом (по схеме для `reverseLeft`):

```
(стартовое_значение /: List())(операция)
```

является эквивалентом (по определению `/:`):

```
стартовое_значение
```

Следовательно, стартовое_значение должно быть `List()`. Чтобы вывести второй операнд, в качестве примера можно взять следующий наименьший список. Поскольку уже известно, что стартовое_значение — это `List()`, можно рассуждать следующим образом:

```
List(x)
```

является эквивалентом (в понятиях свойств `reverseLeft`):

```
reverseLeft(List(x))
```

является эквивалентом (по схеме для `reverseLeft` со стартовым_значением = `List()`):

```
(List() /: List(x)) (операция)
```

является эквивалентом (по определению `/:`):

```
операция (List(), x)
```

Следовательно, операция(`List()`, `x`) является эквивалентом `List(x)`, что можно записать также в виде `x :: List()`. Это наводит на мысль о том, что нужно взять в качестве операции

оператор :: с его операндами, которые поменяли местами. (Иногда такую операцию называют «снок» по отношению к операции ::, именуемый «конс».) И тогда мы приходим к следующей реализации метода reverseLeft:

```
def reverseLeft[T](xs: List[T]) =
  (List[T]() /: xs) {(ys, y) => y :: ys}
```

Чтобы заставить работать механизм вывода типа, здесь также в качестве аннотации типа требуется использовать код List[T](). Если проанализировать вычислительную сложность reverseLeft, можно прийти к выводу, что в нем *n* раз применяется постоянная по времени выполнения операция («снок»), где *n* — длина списка, получаемого в списке аргументов. Таким образом, вычислительная сложность reverseLeft имеет линейную зависимость.

Сортировка списков: sortWith

Операция xs sortWith before, где xs является списком, а before — функцией, которая может использоваться для сравнения двух элементов, выполняет сортировку элементов списка xs. Выражение x before y должно возвращать true, если для намеченной сортировки x должен стоять перед y, например:

```
scala> List(1, -3, 4, 2, 6) sortWith (_ < _)
res51: List[Int] = List(-3, 1, 2, 4, 6)

scala> words sortWith (_.length > _.length)
res52: List[String] = List(quick, brown, the, fox)
```

Обратите внимание на то, что sortWith выполняет сортировку методом слияния подобно тому, как это делает алгоритм msort, показанный в последнем разделе. Но sortWith является методом класса List, а msort определен вне списков.

16.8. Методы объекта List

До сих пор все показанные в этой главе операции реализовывались в качестве методов класса `List`, поэтому они вызывались в отношении отдельно взятых списочных объектов. Существует также ряд методов в глобально доступном объекте `scala.List`, который является объектом-спутником класса `List`. Некоторые из этих операций являются фабричными методами, создающими списки. Другие же — операциями, работающими со списками некоторых конкретных видов. В этом разделе будут представлены обе разновидности методов.

Создание списков из их элементов: List.apply

В книге уже несколько раз попадались литералы списков вида `List(1, 2, 3)`. В их синтаксисе нет ничего особенного. Литерал вида `List(1, 2, 3)` является простым применением объекта `List` к элементам `1`, `2`, `3`. То есть он является эквивалентом кода `List.apply(1, 2, 3)`:

```
scala> List.apply(1, 2, 3)
res53: List[Int] = List(1, 2, 3)
```

Создание диапазона чисел: List.range

Метод `range`, который ранее подробно рассматривался при изучении методов `map` и `flatmap`, создает список, состоящий из диапазона чисел. Его самой простой формой, при которой создаются все числа, начиная с `from` и заканчивая `until` минус один, является `List.range(from, until)`. Следовательно, последнее значение, `until`, в диапазон не входит.

Существует также версия `range`, получающая в качестве третьего параметра значение `step`. В результате выполнения этой операции получится список элементов, следующих друг за другом

с указанным шагом, начиная с `from`. Указываемый шаг `step` может иметь положительное или отрицательное значение:

```
scala> List.range(1, 5)
res54: List[Int] = List(1, 2, 3, 4)

scala> List.range(1, 9, 2)
res55: List[Int] = List(1, 3, 5, 7)

scala> List.range(9, 1, -3)
res56: List[Int] = List(9, 6, 3)
```

Создание однообразных списков: List.fill

Метод `fill` создает список, состоящий из нуля или более копий одного и того же элемента. Он получает два параметра: длину создаваемого списка и повторяемый элемент. Каждый параметр задается в отдельном списке:

```
scala> List.fill(5)('a')
res57: List[Char] = List(a, a, a, a, a)

scala> List.fill(3)("hello")
res58: List[String] = List(hello, hello, hello)
```

Если методу `fill` дать более двух аргументов, он будет создавать многомерные списки, то есть списки списков, списки списков из списков и т. д. Дополнительный аргумент помещается в первый список аргументов.

```
scala> List.fill(2, 3)('b')
res59: List[List[Char]] = List(List(b, b, b),
List(b, b, b))
```

Табулирование функции: List.tabulate

Метод `tabulate` создает список, чьи элементы вычисляются согласно предоставляемой функции. Аргументы у него такие же, как и у метода `List.fill`: в первом списке аргументов задается размерность создаваемого списка, а во втором дается описание элементов списка. Единственным отличием является то, что элементы не фиксируются, а вычисляются из функции:

```
scala> val squares = List.tabulate(5)(n => n * n)
squares: List[Int] = List(0, 1, 4, 9, 16)

scala> val multiplication = List.tabulate(5,5)(_ *
_)
multiplication: List[List[Int]] = List(List(0, 0,
0, 0, 0),
    List(0, 1, 2, 3, 4), List(0, 2, 4, 6, 8),
    List(0, 3, 6, 9, 12), List(0, 4, 8, 12, 16))
```

Объединение нескольких списков: List.concat

Метод `concat` объединяет несколько списков элементов. Объединяемые списки предоставляются `concat` в виде непосредственных аргументов:

```
scala> List.concat(List('a', 'b'), List('c'))
res60: List[Char] = List(a, b, c)

scala> List.concat(List(), List('b'), List('c'))
res61: List[Char] = List(b, c)

scala> List.concat()
res62: List[Nothing] = List()
```

16.9. Совместная обработка нескольких списков

Метод zipped, применяемый в отношении кортежей, обобщает несколько обычных операций для работы не с одним, а сразу с несколькими списками. Одной из таких операций является map. Метод map для двух объединяемых методом zipped списков отображает не отдельные элементы, а пары элементов. Одна пара для первых элементов каждого списка, вторая — для вторых и т. д., то есть количество пар соответствует длине списков. Пример использования выглядит следующим образом:

```
scala> (List(10, 20), List(3, 4, 5)).zipped.map(_
* _)
res63: List[Int] = List(30, 80)
```

Заметьте, что третий элемент второго списка отбрасывается. Метод zipped объединяет только то количество элементов, которое совместно появляется во всех списках. Любые лишние элементы в конце отбрасываются.

Существуют также обобщающие аналоги для методов exists и forall. Они похожи на версии этих методов, предназначенные для работы с одним списком, но работают с элементами не одного, а нескольких списков:

```
scala>      (List("abc",      "de"),      List(3,
2)).zipped.forall(_.length == _)
res64: Boolean = true

scala>      (List("abc",      "de"),      List(3,
2)).zipped.exists(_.length != _)
res65: Boolean = false
```

ускоренный режим чтения

В последнем разделе этой главы дается информация об имеющемся

в Scala алгоритме вывода типа. Если сейчас такие подробности вас не интересуют, можете пропустить весь раздел и сразу перейти к резюме.

16.10. Осмысление имеющегося в Scala алгоритма вывода типа

Одно из отличий предыдущего использования `sortWith` и `msort` касается допустимых синтаксических форм функции сравнения.

Сравните этот диалог с интерпретатором:

```
scala> msort((x: Char, y: Char) => x > y)(abcde)
res66: List[Char] = List(e, d, c, b, a)
```

со следующим:

```
scala> abcde sortWith (_ > _)
res67: List[Char] = List(e, d, c, b, a)
```

Эти два выражения эквивалентны, но в первом используется более длинная форма функции сравнения с именованными параметрами и явно заданными типами. Во втором — более краткая форма, `(_ > _)`, где вместо именованных параметров стоят знаки подчеркивания. Разумеется, с методом `sortWith` вы можете воспользоваться также первой, более длинной формой сравнения.

А вот с `msort` более краткая форма использоваться не может:

```
scala> msort(_ > _)(abcde)
<console>:12: error: missing parameter type for
expanded
function ((x$1, x$2) => x$1.$greater(x$2))
       msort(_ > _)(abcde)
             ^
```

Чтобы понять, почему именно так происходит, следует знать некоторые подробности имеющегося в Scala алгоритма вывода типов. Этот механизм основан на потоках. При использовании метода `m(args)` механизм вывода типов сначала проверяет, имеется ли известный тип у метода `m`. Если такой тип у него есть, именно он и используется для вывода ожидаемого типа аргументов. Например, в выражении `abcde.sortWith(_ > _)` типом `abcde` является `List[Char]`. Таким образом, `sortWith` известен как метод, получающий аргумент типа `(Char, Char) => Boolean` и выдающий результат типа `List[Char]`. Поскольку типы параметров аргументов функции известны, их не нужно записывать явным образом. По совокупности всего известного о методе `sortWith` механизм вывода типов может установить, что код `(_ > _)` нужно раскрыть в `((x: Char, y: Char) => x > y)`, где `x` и `y` — некие произвольные только что полученные имена.

Теперь рассмотрим второй вариант, `msort(_ > _)(abcde)`. Типом `msort` является каррированный полиморфный тип метода, который забирает аргумент типа `(T, T) => Boolean` в функцию из `List[T]` в `List[T]`, где `T` — некий пока еще неизвестный тип. Прежде чем он будет применен к своим аргументам, у метода `msort` должен быть создан экземпляр с параметром типа.

Поскольку точный тип экземпляра `msort` в приложении еще неизвестен, он не может быть использован для вывода типа своего первого аргумента. В этом случае механизм вывода типов меняет свою стратегию: сначала он проверяет тип аргументов метода для определения экземпляра метода с подходящим типом. Но, когда перед ним стоит задача проверки типа функционального литерала в краткой форме записи, `(_ > _)`, он дает сбой из-за отсутствия информации о подразумеваемых типах заданных параметров функции, показанных знаками подчеркивания.

Одним из способов решения проблемы является передача `msort` явно заданного типа параметра:

```
scala> msort[Char](_ > _)(abcde)
res68: List[Char] = List(e, d, c, b, a)
```

Поскольку экземпляр `msort` подходящего типа теперь известен, он может быть использован для вывода типов аргументов. Еще одним возможным решением может стать перезапись метода `msort` таким образом, чтобы его параметры поменялись местами:

```
def msortSwapped[T](xs: List[T])(less:
    (T, T) => Boolean): List[T] = {

  // Та же реализация, что и у msort,
  // но с аргументами, которые поменялись местами
}
```

Теперь вывод типов будет выполнен успешно:

```
scala> msortSwapped(abcde)(_ > _)
res69: List[Char] = List(e, d, c, b, a)
```

Получилось так, что механизм вывода типа воспользовался известным типом первого параметра `abcde` для определения параметра типа метода `msortSwapped`. Поскольку точный тип `msortSwapped` был известен, то он может быть использован для вывода типа второго параметра, `(_ > _)`.

В общем, когда ставится задача вывести параметры типа полиморфного метода, механизм вывода типов принимает во внимание типы всех значений аргументов в первом списке параметров, игнорируя все аргументы, кроме этих. Поскольку `msortSwapped` является каррированным методом с двумя списками параметров, обращать внимание на второй аргумент (то есть на функциональное значение) для определения параметра типа метода не нужно.

Эта схема вывода типа предлагает следующий принцип

разработки библиотек: при создании полиморфного метода, получающего некие нефункциональные аргументы и функциональный аргумент, этот функциональный аргумент в самом каррированном списке параметров нужно поставить на последнее место. Тогда экземпляр метода подходящего типа может быть выведен из нефункциональных аргументов, и этот тип в свою очередь может быть использован для проверки типа функционального аргумента. Совокупный эффект будет состоять в том, что пользователи метода получат возможность предоставления меньшего объема информации и написания функциональных литералов более компактными способами.

Теперь рассмотрим более сложный случай, касающийся операции *свертки*. В чем необходимость явного указания параметра типа в выражении, подобном телу показанного здесь метода `flattenRight`?

```
(xss :\ List[T]()) (_ ::: _)
```

Тип операции правой свертки является полиморфным в двух переменных типа. Если взять выражение

```
(xs :\ z) (op)
```

то типом `xs` должен быть список какого-то произвольного типа `A`, скажем `xs: List[A]`. Начальное значение `z` может быть какого-нибудь другого типа `B`. Тогда операция `op` должна получать два аргумента типа, `A` и `B`, и возвращать результат типа `B`, то есть `op: (A, B) => B`. Поскольку тип значения `z` не связан с типом списка `xs`, у механизма вывода типов нет контекстной информации для `z`.

Теперь рассмотрим выражение в ошибочной версии метода `flattenRight`, также показанное здесь:

```
(xss :\ List()) (_ ::: _) // этот код не пройдет компиляцию
```

Начальное значение `z` в этой свертке является пустым списком, `List()`, следовательно, без дополнительной информации о типе его тип будет выведен как `List[Nothing]`. Исходя из этого механизм вывода типов установит, что типом `B` в свертке будет являться `List[Nothing]`. Таким образом, для операции `(_ ::: _)` в свертке будет ожидаться следующий тип:

```
(List[T], List[Nothing]) => List[Nothing]
```

Конечно же, такой тип возможен для операции в данной свертке, но пользы от него никакой! Он сообщает, что операция всегда получает в качестве второго аргумента пустой список и всегда в качестве результата выдает также пустой список.

Иными словами, вопрос вывода типа на основе `List()` был решен слишком рано — он должен был выждать, пока не станет виден тип операции `op`. Следовательно, весьма полезное в иных случаях правило о том, что для определения типа метода нужно принимать во внимание только первый раздел аргументов, при применении его к каррированному методу становится камнем преткновения. В то же время, даже если бы это правило было смягчено, механизм вывода типов все равно не смог бы определиться с типом для операции `op`, поскольку ее типы параметров не приведены. Таким образом, создается ситуация, как в уловке-22, которую можно разрешить путем явной аннотации типа, получаемой от программиста.

Данный пример выявляет ряд ограничений локальной, основанной на потоках схемы вывода типов, имеющейся в Scala. В более глобальном механизме вывода типов в стиле Хиндли — Милнера (Hindley — Milner), используемом в таких функциональных языках, как ML или Haskell, подобных ограничений нет. Но по сравнению со стилем Хиндли — Милнера имеющийся в Scala механизм вывода типов обходится с объектно-ориентированной системой подтипов намного изящнее. К счастью, ограничения проявляются только в некоторых крайних случаях и

обычно их без особого труда можно обойти за счет добавления явной аннотации типа.

Добавление аннотаций типа пригодится также при отладке, когда вы будете поставлены в тупик сообщениями об ошибках типа, связанными с полиморфными методами. Если нет уверенности в причине возникновения конкретной ошибки типа, нужно просто добавить некоторые аргументы типа или другие аннотации типа, в правильности которых вы не сомневаетесь. Тогда можно будет быстро понять, где реальный источник проблемы.

Резюме

В этой главе были показаны многие способы работы со списками. Рассмотрены основные операции, такие как `head` и `tail`, операции первого порядка, такие как `reverse`, операции высшего порядка, такие как `map`, а также полезные методы, определенные в объекте `List`. Попутно были изучены принципы работы имеющегося в Scala механизма вывода типов.

Списки в Scala являются настоящей рабочей лошадкой, поэтому, узнав, как с ними работать, вы сможете извлечь для себя немалую выгоду. Но списки являются всего лишь одной из разновидностей коллекций, поддерживаемых в Scala. Тематика следующей главы будет скорее охватывающей, чем углубленной. В ней мы покажем порядок применения различных типов коллекций, имеющихся в Scala.

97 Графическое представление структуры списка типа List показано на рис. 22.2.

98 Более подробно ковариантность и другие разновидности вариаций рассмотрены в главе 19.

99 Более подробно параметры типов будут рассмотрены в главе 19.

100 Метод ???, выдающий ошибку scala.NotImplementedError и имеющий тип результата Nothing, может применяться в качестве временной реализации в процессе разработки приложения.

101 Как уже упоминалось в разделе 10.12, понятие пары является неформальным названием для Tuple2.

102 Имеется в виду класс scala.StringBuilder, а не java.lang.StringBuilder.

103 Под операторами высшего порядка понимаются функции высшего порядка, используемые в системе записи операторов. Как упоминалось в разделе 9.1, функция относится к высшему порядку, если получает в качестве параметров одну функцию и более.

17. Работа с другими коллекциями

В Scala имеется весьма богатая библиотека коллекций. В этой главе последовательно рассматриваются наиболее востребованные типы коллекций и проводимые над ними операции. Более полный обзор доступных коллекций будет представлен в главе 24, а в главе 25 показано, как композиционные конструкции Scala используются для предоставления насыщенного API.

17.1. Последовательности

Типы последовательностей позволяют работать с группами данных, выстроенных по порядку. Поскольку элементы упорядочены, можно запрашивать первый элемент, второй, 103-й и т. д. В этом разделе будет предоставлен беглый обзор наиболее важных последовательностей.

Списки

Возможно, самым важным типом последовательности, о котором следует знать, является класс `List` — неизменяемый связанный список, который подробно рассмотрен в предыдущей главе. Списки поддерживают быстрое добавление и удаление элементов в начале списка, но не обеспечивают возможности получить быстрый доступ к произвольным индексам, поскольку реализация вынуждена выполнять последовательный обход всех элементов списка.

Это сочетание свойств может показаться странным, но оно попало в золотую середину и неплохо работает во многих алгоритмах. Как следует из описаний, представленных в главе 15, быстрое добавление и удаление начальных элементов означает хорошую работу поиска по шаблону. Неизменяемость списков помогает разрабатывать корректные эффективные алгоритмы,

поскольку избавляет от необходимости создания копий списков.

Краткий пример, показывающий способ инициализации списка и получения доступа к его голове и хвосту, выглядит следующим образом:

```
scala> val colors = List("red", "blue", "green")
colors: List[String] = List(red, blue, green)

scala> colors.head
res0: String = red

scala> colors.tail
res1: List[String] = List(blue, green)
```

Чтобы освежить в памяти сведения о списках, обратитесь к шагу 8 главы 3. А подробности использования списков можно найти в главе 16. Списки будут рассматриваться также в главе 22, которая дает представление о том, как именно они реализованы в Scala.

Массивы

Массивы позволяют хранить последовательность элементов и оперативно обращаться к элементу, находящемуся в произвольной позиции, с целью либо получения этого элемента, либо его обновления, для чего используется индекс, отсчитываемый от нуля. Массив известной длины, для которого пока неизвестны значения элементов, создается следующим образом:

```
scala> val fiveInts = new Array[Int](5)
fiveInts: Array[Int] = Array(0, 0, 0, 0, 0)
```

А вот как инициализируется массив, когда значения элементов известны:

```
scala> val fiveToOne = Array(5, 4, 3, 2, 1)
```

```
fiveToOne: Array[Int] = Array(5, 4, 3, 2, 1)
```

Как уже упоминалось, доступ к массивам в Scala осуществляется указанием индекса в круглых, а не в квадратных, как в Java, скобках. Рассмотрим пример доступа к элементу массива и обновления элемента:

```
scala> fiveInts(0) = fiveToOne(4)

scala> fiveInts
res3: Array[Int] = Array(1, 0, 0, 0, 0)
```

Массивы в Scala представлены точно так же, как массивы в Java. Поэтому можно абсолютно свободно использовать имеющиеся в Java методы, возвращающие массивы[104].

В предыдущих главах действия с массивами встречались уже много раз. Основы этих действий были рассмотрены в шаге 7 главы 3. Ряд примеров поэлементного обхода массивов с применением выражения `for` был показан в разделе 7.3. Массивы также занимают видное место в библиотеке двумерной разметки в главе 10.

Списочный буфер

Класс `List` предоставляет быстрый доступ к голове и хвосту списка, но не к его концу. Таким образом, если нужно построить список с добавлением элементов к концу, следует рассматривать возможность построения списка в обратном порядке путем добавления элементов спереди. Затем, когда это будет сделано, нужно вызвать метод реверсирования `reverse`, чтобы получить элементы в нужном порядке.

Другим вариантом, позволяющим избежать реверсирования, является использование объекта `ListBuffer`. Это содержащийся в пакете `scala.collection.mutable` изменяемый объект,

который может помочь более эффективно строить списки, когда нужно добавлять элементы к их концу. `ListBuffer` обеспечивает постоянное время выполнения операций добавления элементов как к концу, так и к началу списка. К концу списка элемент добавляется с помощью оператора `+=`, а к началу — с помощью оператора `+=:`. Когда построение будет завершено, можно получить список типа `List` путем вызова в отношении `ListBuffer` метода `toList`. Соответствующий пример выглядит так:

```
scala> import scala.collection.mutable.ListBuffer
import scala.collection.mutable.ListBuffer

scala> val buf = new ListBuffer[Int]
buf:   scala.collection.mutable.ListBuffer[Int]   =
ListBuffer()

scala> buf += 1
res4: buf.type = ListBuffer(1)

scala> buf += 2
res5: buf.type = ListBuffer(1, 2)

scala> buf
res6: scala.collection.mutable.ListBuffer[Int] =

ListBuffer(1, 2)


scala> 3 +=: buf
res7: buf.type = ListBuffer(3, 1, 2)

scala> buf.toList
res8: List[Int] = List(3, 1, 2)
```

Еще одним поводом для использования ListBuffer вместо List является предотвращение потенциальной возможности переполнения стека. Если есть возможность создания списка в нужном порядке путем добавления элементов в его начало, но рекурсивный алгоритм, который потребуется, не является алгоритмом с концевой рекурсией, можно вместо этого воспользоваться выражением for или циклом while и ListBuffer. Такой способ применения ListBuffer будет показан в разделе 22.2.

Буфер массива

Объект ArrayBuffer похож на массив, за исключением того, что в дополнение ко всему здесь предоставляется возможность добавления и удаления элементов в начале и в конце последовательности. Доступны все те же операции, что и в классе Array, хотя они выполняются несколько медленнее из-за наличия в реализации уровня-оболочки.

Чтобы воспользоваться ArrayBuffer, нужно сначала импортировать его из пакета изменяемых коллекций:

```
scala> import scala.collection.mutable.ArrayBuffer
import scala.collection.mutable.ArrayBuffer
```

При создании ArrayBuffer нужно задать параметр типа, а длину указывать необязательно. По мере надобности ArrayBuffer автоматически установит выделяемое пространство памяти:

```
scala> val buf = new ArrayBuffer[Int]()
buf: scala.collection.mutable.ArrayBuffer[Int] =
   ArrayBuffer()
```

Добавить элемент к ArrayBuffer можно с помощью метода +=:

```
scala> buf += 12
res9: buf.type = ArrayBuffer(12)

scala> buf += 15
res10: buf.type = ArrayBuffer(12, 15)

scala> buf
res11: scala.collection.mutable.ArrayBuffer[Int] =
    ArrayBuffer(12, 15)
```

Доступны все обычные методы работы с массивами. Например, можно запросить у `ArrayBuffer` его длину или извлечь элемент по его индексу:

```
scala> buf.length
res12: Int = 2

scala> buf(0)
res13: Int = 12
```

Строки, реализуемые посредством StringOps

Еще одной последовательностью, заслуживающей упоминания, является `StringOps`. В ней реализованы многие методы работы с последовательностями. Поскольку в `Predef` имеется подразумеваемое преобразование из `String` в `StringOps`, в качестве последовательности можно рассматривать любую строку. Вот пример:

```
scala> def hasUpperCase(s: String) =
s.exists(_.isUpper)
hasUpperCase: (s: String)Boolean
scala> hasUpperCase("Robert Frost")
res14: Boolean = true
```

```
scala> hasUpperCase("e e cummings")
res15: Boolean = false
```

В этом примере метод `exists` вызывается в отношении строки, которая в теле метода `hasUpperCase` назвается `s`. Поскольку в самом классе `String` никакого метода по имени `exists` не объявлено, компилятор Scala выполнит подразумеваемое преобразование `s` в `StringOps`, где такой метод имеется. Метод `exists` считает строку последовательностью символов, и, если любой из символов относится к верхнему регистру, им будет возвращено значение `true`[105].

17.2. Наборы и отображения

В предыдущих главах, начиная с шага 10 главы 3, уже были показаны основы наборов и отображений. Прочитав этот раздел, вы получите более глубокое представление об их использовании и сможете рассмотреть несколько дополнительных примеров.

Как уже упоминалось, библиотека коллекций Scala предлагает как изменяемые, так и неизменяемые версии наборов и отображений. Иерархия наборов показана на рис. 3.2, а иерархия отображений — на рис. 3.3. Из этих диаграмм следует, что простые имена `Set` и `Map` используются тремя трейтами и все они находятся в разных пакетах.

По умолчанию, когда в коде применяется `Set` или `Map`, вы получаете неизменяемый объект. Если нужен изменяемый вариант, следует воспользоваться явно указанным импортированием. К неизменяемым вариантам Scala предоставляет самый простой доступ — в качестве небольшого поощрения за то, что предпочтение отдано им, а не их изменяемым аналогам. Доступ предоставляется через объект `Predef`, подразумеваемо импортируемый в каждый файл исходного кода на языке Scala. Соответствующие определения

показаны в листинге 17.1.

Листинг 17.1. Исходные определения отображений map и наборов set в объекте Predef

```
object Predef {
  type Map[A, +B] = collection.immutable.Map[A, B]
  type Set[A] = collection.immutable.Set[A]
  val Map = collection.immutable.Map
  val Set = collection.immutable.Set
  // ...
}
```

Для определения `Set` и `Map` в качестве псевдонимов для более длинных полных имен трейтов неизменяемых наборов и отображений в `Predef` использовано ключевое слово `type`[106]. Чтобы ссылаться на синглтон-объекты для неизменяемых `Set` и `Map`, выполняется инициализация `val`-переменных с именами `Set` и `Map`. Следовательно, `Map` является тем же, что и объект `Predef.Map`, который определен быть тем же самым, что и `scala.collection.immutable.Map`. Это справедливо как для типа `Map`, так и для `Map`-объекта.

Если нужно воспользоваться как изменяемыми, так и неизменяемыми наборами или отображениями, то одним из подходов является импортирование имен пакетов, в которых содержатся изменяемые варианты:

```
scala> import scala.collection.mutable
import scala.collection.mutable
```

Можно продолжать ссылаться на неизменяемый набор, как и прежде `Set`, но теперь можно будет сослаться и на изменяемый набор, указав `mutable.Set`. Вот как выглядит соответствующий

пример:

```
scala> val mutaSet = mutable.Set(1, 2, 3)
mutaSet:    scala.collection.mutable.Set[Int]    =
Set(1, 2, 3)
```

Использование наборов

Основной характеристикой наборов является гарантия наличия в списке только уникальных, по определению оператора ==, объектов. В качестве примера воспользуемся набором, чтобы вычислить имеющиеся в строке уникальные слова.

Метод split класса String, если указать в качестве разделителей слов пробелы и знаки пунктуации, может разбить строку на слова. Для этого вполне достаточно будет применить регулярное выражение [!,.]+: оно показывает, что строка должна быть разбита во всех местах, где имеется один или несколько пробелов и/или знак пунктуации.

```
scala> val text = "See Spot run. Run, Spot. Run!"
text: String = See Spot run. Run, Spot. Run!

scala> val wordsArray = text.split("[ !,.]+")
wordsArray: Array[String]
  = Array(See, Spot, run, Run, Spot, Run)
```

Чтобы вычислить уникальные слова, их можно преобразовать, приведя их символы к единому регистру, а затем добавить к набору. Поскольку наборы исключают дубликаты, каждое уникальное слово будет появляться в наборе только один раз.

Сначала можно создать пустой набор, используя метод empty, предоставляемый объектом-спутником Set:

```
scala> val words = mutable.Set.empty[String]
```

```
words:    scala.collection.mutable.Set[String]    =
Set()
```

Затем, просто перебирая слова с помощью выражения `for`, можно преобразовать каждое слово, приведя его символы к нижнему регистру, а после чего добавить его к изменяемому набору с помощью оператора `+=`:

```
scala> for (word <- wordsArray)
words += word.toLowerCase

scala> words
res17:    scala.collection.mutable.Set[String]    =
Set(see, run, spot)
```

Таким образом, в тексте содержатся три уникальных слова: `spot`, `run` и `see`. Наиболее часто используемые методы, применяемые как к изменяемым, так и к неизменяемым наборам, показаны в табл. 17.1.

Таблица 17.1. Наиболее распространенные операторы для работы с наборами

Что используется	Что этот метод делает
val nums = Set(1, 2, 3)	Создает неизменяемый набор (nums.toString возвращает Set(1, 2, 3))
nums + 5	Добавляет элемент (возвращает Set(1, 2, 3, 5))
nums – 3	Удаляет элемент (возвращает Set(1, 2))
nums ++ List(5, 6)	Добавляет несколько элементов (возвращает Set(1, 2, 3, 5, 6))
nums –– List(1, 2)	Удаляет несколько элементов (возвращает Set(3))
nums & Set(1, 3, 5, 7)	Берет пересечение двух наборов (возвращает Set(1, 3))
nums.size	Возвращает размер набора (возвращает 3)
nums.contains(3)	Проверка включения (возвращает true)
import scala.collection.mutable	Упрощает доступ к изменяемым коллекциям
val words =	Создает пустой, изменяемый набор (words.toString

mutable.Set.empty[String]	возвращает Set())
words += "the"	Добавляет элемент (words.toString возвращает Set(the))
words –= "the"	Удаляет элемент, если он существует (words.toString возвращает Set())
words ++= List("do", "re", "mi")	Добавляет несколько элементов (words.toString возвращает Set(do, re, mi))
words ––= List("do", "re")	Удаляет несколько элементов (words.toString возвращает Set(mi))
words.clear	Удаляет все элементы (words.toString возвращает Set())

Применение отображений

Отображения позволяют связать значение с каждым элементом набора. Использование отображений похоже на применение массива, за исключением того, что вместо индексирования с помощью целых чисел, начинающихся с нуля, можно воспользоваться ключами любого вида. Если импортировать пакет с именем `mutable`, можно создать пустое изменяемое отображение:

```
scala> val map = mutable.Map.empty[String, Int]
map: scala.collection.mutable.Map[String,Int] = Map()
```

Учтите, что при создании отображения следует указать два типа. Первый тип предназначен для *ключей* отображения, а второй — для их *значений*. В данном случае ключами являются строки, а значениями — целые числа. Задание записей в отображении похоже на задание записей в массиве:

```
scala> map("hello") = 1

scala> map("there") = 2

scala> map
```

```
res20: scala.collection.mutable.Map[String,Int] =
    Map(hello -> 1, there -> 2)
```

По аналогии с этим чтение отображения похоже на чтение массива:

```
scala> map("hello")
res21: Int = 1
```

Чтобы связать все воедино, рассмотрим метод, подсчитывающий количество появлений каждого из слов в строке:

```
scala> def countWords(text: String) = {
            val counts = mutable.Map.empty[String,
Int]
         for (rawWord <- text.split("[ ,!.]+")) {
           val word = rawWord.toLowerCase
           val oldCount =
                       if (counts.contains(word))
counts(word)
             else 0
           counts += (word -> (oldCount + 1))
         }
         counts
       }
countWords: (text:
String)scala.collection.mutable.Map[String,Int]

scala> countWords("See Spot run! Run, Spot. Run!")
res22: scala.collection.mutable.Map[String,Int] =
    Map(spot -> 2, see -> 1, run -> 3)
```

По результатам подсчетов можно увидеть, что в тексте много говорится про бег (`run`) и совсем мало — про наблюдение (`see`).

Этот код работает благодаря использованию изменяемого отображения по имени `counts` и отображению каждого слова на количество его появлений в тексте. Для каждого слова в тексте выполняются поиск предыдущего подсчета количества появлений слова и его увеличение на единицу, а затем в `counts` сохраняется новое значение подсчета. Обратите внимание: для проверки того, встречалось это слово раньше или нет, используется метод `contains`. Если `counts.contains(word)` не возвращает `true`, значит слово еще не встречалось и для подсчета используется нуль.

Многие из наиболее часто используемых методов работы как с изменяемыми, так и с неизменяемыми отображениями показаны в табл. 17.2.

Таблица 17.2. Наиболее часто используемые операции для работы с отображениями

Что используется	Что этот метод делает
val nums = Map("i" –> 1, "ii" –> 2)	Создает неизменяемое отображение (nums.toString возвращает Map(i –> 1, ii –> 2))
nums + ("vi" –> 6)	Добавляет запись (возвращает Map(i –> 1, ii –> 2, vi –> 6))
nums – "ii"	Удаляет запись (возвращает Map(i –> 1))
nums ++ List("iii" –> 3, "v" –> 5)	Добавляет несколько записей (возвращает Map(i –> 1, ii –> 2, iii –> 3, v –> 5))
nums –– List("i", "ii")	Удаляет несколько записей (возвращает Map())
nums.size	Возвращает размер отображения (возвращает 2)
nums.contains("ii")	Проверяет присутствие (возвращает true)
nums("ii")	Извлекает значение по указанному ключу (возвращает 2)
nums.keys	Возвращает ключи (возвращает результат итерации, выполненной над строками "i" и "ii")
nums.keySet	Возвращает ключи в виде набора (возвращает Set(i, ii))
nums.values	Возвращает значения (возвращает результат итерации, выполненной над целыми числами 1 и 2)
nums.isEmpty	Показывает, является ли отображение пустым (возвращает false)
import scala.collection.mutable	Упрощает доступ к изменяемым коллекциям

val words = mutable.Map.empty[String, Int]	Создает пустое изменяемое отображение
words += ("one" –> 1)	Добавляет запись в отображение из ключа "one" и значения 1 (words.toString возвращает Map(one –> 1))
words –= "one"	Удаляет запись из отображения, если она существует (words.toString возвращает Map())
words ++= List("one" –> 1, "two" –> 2, "three" -> 3)	Добавляет записи в изменяемое отображение (words.toString возвращает Map(one –> 1, two –> 2, three –> 3))
words –-= List("one", "two")	Удаляет несколько объектов (words.toString возвращает Map(three –> 3))

Наборы и отображения, используемые по умолчанию

Для большинства случаев реализаций изменяемых и неизменяемых наборов и отображений, предоставляемых `Set()`, `scala.collection.mutable.Map()` и тому подобными фабриками, наверное, вполне достаточно. Реализации, предоставляемые этими фабриками, используют алгоритм ускоренного поиска, в котором обычно задействуется хеш-таблица, поэтому они могут быстро обнаружить присутствие или отсутствие объекта в коллекции.

Фабричный метод `scala.collection.mutable.Set()`, к примеру, возвращает `scala.collection.mutable.HashSet`, внутри которого используется хеш-таблица. Аналогично этому фабричный метод `scala.collection.mutable.Map()` возвращает `scala.collection.mutable.HashMap`.

История для неизменяемых наборов и отображений складывается несколько сложнее. Как показано в табл. 17.3, класс, возвращенный фабричным методом `scala.collection.immutable.Set()`, зависит, к примеру, от того, сколько элементов ему было передано. С целью достижения максимальной производительности для наборов любого размера, состоящих не более чем из пяти элементов, используется специальный класс. Но при запросе набора из пяти и более

элементов фабричный метод вернет реализацию, использующую хеш-извлечения.

По аналогии с этим, как следует из табл. 17.3, в результате выполнения фабричного метода `scala.collection.immutable.Map()` будет возвращен нужный класс в зависимости от того, сколько пар «ключ — значение» ему передано. Как и в случае с наборами, для того чтобы неизменяемые отображения с количеством элементов меньше пяти достигли максимальной производительности для отображения каждого конкретного размера, используется специальный класс. Но, если отображение содержит пять и более пар «ключ — значение», используется неизменяемый класс `HashMap`.

Для обеспечения максимальной производительности используемые по умолчанию реализации неизменяемых классов, показанные в табл. 17.3 и 17.4, работают совместно. Например, когда добавляется элемент к `EmptySet`, возвращается `Set1`. Если добавляется элемент к этому `Set1`, возвращается `Set2`. Если затем удалить элемент из `Set2`, будет опять получен `Set1`.

Таблица 17.3. Реализации используемых по умолчанию неизменяемых наборов

Количество элементов	Реализация
0	scala.collection.immutable.EmptySet
1	scala.collection.immutable.Set1
2	scala.collection.immutable.Set2
3	scala.collection.immutable.Set3
4	scala.collection.immutable.Set4
5 или более	scala.collection.immutable.HashSet

Таблица 17.4. Реализации используемых по умолчанию неизменяемых отображений

Количество элементов	Реализация
0	scala.collection.immutable.EmptyMap
1	scala.collection.immutable.Map1
2	scala.collection.immutable.Map2
3	scala.collection.immutable.Map3
4	scala.collection.immutable.Map4
5 или более	scala.collection.immutable.HashMap

Отсортированные наборы и отображения

Иногда может понадобиться набор или отображение, чей итератор возвращает элементы в определенном порядке. Для этого в библиотеке коллекций Scala имеются трейты `SortedSet` и `SortedMap`. Они реализованы с использованием классов `TreeSet` и `TreeMap`, которые для хранения элементов в определенном порядке применяют красно-черное дерево (в данном случае `TreeSet`) или ключи (в случае с `TreeMap`). Порядок определяется трейтом `Ordered`, чей тип элементов набора или тип ключей отображения должен быть либо подмешан, либо пропущен через механизм подразумеваемого преобразования. Эти классы поставляются только в неизменяемых вариантах. Рассмотрим ряд примеров использования `TreeSet`:

```
scala> import scala.collection.immutable.TreeSet
import scala.collection.immutable.TreeSet

scala> val ts = TreeSet(9, 3, 1, 8, 0, 2, 7, 4, 6,
5)
ts: scala.collection.immutable.TreeSet[Int] =
    TreeSet(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> val cs = TreeSet('f', 'u', 'n')
cs: scala.collection.immutable.TreeSet[Char] =
```

```
    TreeSet(f, n, u)
```

А это ряд примеров использования TreeMap:

```
scala> import scala.collection.immutable.TreeMap
import scala.collection.immutable.TreeMap

scala> var tm = TreeMap(3 -> 'x', 1 -> 'x', 4 ->
'x')
tm: scala.collection.immutable.TreeMap[Int,Char] =
    Map(1 -> x, 3 -> x, 4 -> x)

scala> tm += (2 -> 'x')

scala> tm
res30:
scala.collection.immutable.TreeMap[Int,Char] =
    Map(1 -> x, 2 -> x, 3 -> x, 4 -> x)
```

17.3. Какие коллекции выбрать: изменяемые или неизменяемые

При решении одних задач хорошо работают изменяемые коллекции, а при решении других — неизменяемые. В случае сомнений лучше начать с неизменяемой коллекции, а позже при необходимости перейти к изменяемой, поскольку разобраться в работе неизменяемых коллекций гораздо проще.

Иногда может оказаться полезно двигаться в обратном направлении. Когда код, использующий изменяемые коллекции, становится сложным и в нем трудно разобраться, следует подумать, а не поможет ли замена некоторых коллекций их неизменяемыми альтернативами. В частности, если волнует вопрос создания копий изменяемых коллекций только в нужных

местах или приходится слишком сильно задумываться над тем, кто владеет изменяемой коллекцией или что именно она содержит, есть смысл обратиться к переключению некоторых коллекций на их неизменяемые аналоги.

Кроме того что в потенциально неизменяемых коллекциях легче разобраться, они, как правило, при небольшом количестве хранящихся в них элементов могут потребовать для хранения меньше места, чем их изменяемые аналоги. Например, пустое изменяемое отображение в его получаемом по умолчанию представлении в виде `HashMap` занимает примерно 80 байтов и около 16 дополнительных байтов требуется для добавления к нему каждой записи. А пустое неизменяемое отображение `Map` является одним объектом, который совместно используется всеми ссылками, причем ссылка на него обходится, по сути, в одно поле указателя.

Более того, в настоящее время библиотека коллекций Scala хранит до четырех записей неизменяемых отображений и наборов в одном объекте, который в зависимости от количества хранящихся в коллекции записей обычно занимает от 16 до 40 байтов[107]. Следовательно, для небольших наборов и отображений неизменяемые версии занимают намного меньше места, чем изменяемые. С учетом того что многие коллекции имеют небольшой размер, перевод их в разряд неизменяемых может принести существенную экономию пространства памяти и выгодные условия для повышения производительности системы.

Чтобы облегчить переход с неизменяемых на изменяемые коллекции и наоборот, в Scala предоставляются некоторые синтаксические изыски. Хотя неизменяемые наборы и отображения не поддерживают настоящий метод `+=`, в Scala дается полезная альтернативная интерпретация `+=`. Когда используется запись `a += b` и `a` не поддерживает метод по имени `+=`, Scala пытается интерпретировать эту запись как `a = a + b`.

Например, неизменяемые наборы не поддерживают оператор

+=:

```
scala> val people = Set("Nancy", "Jane")
people: scala.collection.immutable.Set[String] =
    Set(Nancy, Jane)

scala> people += "Bob"
<console>:14: error: value += is not a member of
scala.collection.immutable.Set[String]
              people += "Bob"
                     ^
```

Но если объявить `people` в качестве `var`-, а не `val`-переменной, то коллекция может быть обновлена с помощью операции `+=` даже при том, что она неизменяемая. Сначала создается новая коллекция, а затем переменной `people` присваивается новое значение для ссылки на новую коллекцию:

```
scala> var people = Set("Nancy", "Jane")

people: scala.collection.immutable.Set[String] =
    Set(Nancy, Jane)

scala> people += "Bob"
scala> people
res34: scala.collection.immutable.Set[String] =
    Set(Nancy, Jane, Bob)
```

После этой серии инструкций переменная `people` ссылается на новый неизменяемый набор, содержащий добавленную строку `"Bob"`. Та же идея применима не только к методу `+=`, но и к любому другому методу, заканчивающемуся знаком `=`. Вот как тот же самый синтаксис используется с оператором `-=`, который удаляет элемент из набора, и с оператором `++=`, добавляющим к

набору коллекцию элементов:

```
scala> people -= "Jane"

scala> people ++= List("Tom", "Harry")

scala> people
res37: scala.collection.immutable.Set[String] =
    Set(Nancy, Bob, Tom, Harry)
```

Чтобы понять, насколько это полезно, рассмотрим еще раз следующий пример отображения Map из раздела 1.1:

```
var capital = Map("US" -> "Washington", "France" -
> "Paris")
capital += ("Japan" -> "Tokyo")
println(capital("France"))
```

В этом коде используются неизменяемые коллекции. Если захочется попробовать воспользоваться вместо них изменяемыми коллекциями, нужно будет всего лишь импортировать изменяемую версию Map, переопределив таким образом выполненный по умолчанию импорт неизменяемой версии Map:

```
import   scala.collection.mutable.Map          //
единственное требуемое изменение!
var capital = Map("US" -> "Washington", "France" -
> "Paris")
capital += ("Japan" -> "Tokyo")
println(capital("France"))
```

Так легко удастся преобразовать не все примеры, но особая трактовка методов, заканчивающихся знаком равенства, зачастую сокращает объем кода, требующего изменений.

Кстати, эта трактовка синтаксиса работает не только с

коллекциями, но и с любыми разновидностями значений. Например, здесь она была использована в отношении чисел с плавающей точкой:

```
scala> var roughlyPi = 3.0
roughlyPi: Double = 3.0

scala> roughlyPi += 0.1

scala> roughlyPi += 0.04

scala> roughlyPi
res40: Double = 3.14
```

Эффект от такого распространения похож на эффект, получаемый от операторов присваивания, использующихся в Java (+=, -=, *= и т. п.), но он носит более общий характер, поскольку преобразован может быть каждый оператор, оканчивающийся на =.

17.4. Инициализация коллекций

Как уже было показано, наиболее широко востребованным способом создания и инициализации коллекции является передача исходных элементов фабричному методу, определенному в объекте-спутнике класса выбранной вами коллекции. Элементы просто помещаются в круглые скобки после имени объекта-спутника, и компилятор Scala приступает в отношении этого объекта-спутника к преобразованию кода в вызов метода `apply`:

```
scala> List(1, 2, 3)
res41: List[Int] = List(1, 2, 3)

scala> Set('a', 'b', 'c')
res42:    scala.collection.immutable.Set[Char]    =
```

```
Set(a, b, c)

scala> import scala.collection.mutable
import scala.collection.mutable

scala> mutable.Map("hi" -> 2, "there" -> 5)
res43: scala.collection.mutable.Map[String,Int] =
    Map(hi -> 2, there -> 5)

scala> Array(1.0, 2.0, 3.0)
res44: Array[Double] = Array(1.0, 2.0, 3.0)
```

Чаще всего можно позволить компилятору Scala вывести тип элемента коллекции из элементов, переданных ее фабричному методу, но иногда может понадобиться создать коллекцию, указав при этом тип, отличающийся от того, который выберет компилятор. Это особенно касается изменяемых коллекций. Рассмотрим пример:

```
scala> import scala.collection.mutable
import scala.collection.mutable

scala> val stuff = mutable.Set(42)
stuff: scala.collection.mutable.Set[Int] = Set(42)

scala> stuff += "abracadabra"
<console>:16: error: type mismatch;
found : String("abracadabra")
required: Int
              stuff += "abracadabra"
                       ^
```

Проблема здесь заключается в том, что переменной `stuff` был

получен тип элемента Int. Если нужно, чтобы имелся тип элемента Any, об этом следует заявить явно, поместив тип элемента в квадратные скобки:

```
scala> val stuff = mutable.Set[Any](42)
stuff: scala.collection.mutable.Set[Any] = Set(42)
```

Еще одна особая ситуация возникает при желании инициализировать коллекцию с помощью другой коллекции. Представим, к примеру, что имеется список, но нужно получить коллекцию TreeSet, содержащую элементы, находящиеся в списке. Список выглядит следующим образом:

```
scala> val colors = List("blue", "yellow", "red",
"green")
colors: List[String] = List(blue, yellow, red,
green)
```

Передать список названий цветов фабричному методу для TreeSet невозможно:

```
scala> import scala.collection.immutable.TreeSet
import scala.collection.immutable.TreeSet

scala> val treeSet = TreeSet(colors)
<console>:16: error: No implicit Ordering defined
for
List[String].
       val treeSet = TreeSet(colors)
                             ^
```

Вместо этого потребуется создать пустую коллекцию TreeSet[String] и добавить к ней элементы списка с использованием имеющегося в классе TreeSet оператора ++:

```
scala> val treeSet = TreeSet[String]() ++ colors
treeSet:
scala.collection.immutable.TreeSet[String] =
    TreeSet(blue, green, red, yellow)
```

Преобразование в массив или список

А вот если нужно инициализировать список или массив с помощью другой коллекции, задача решается довольно просто. Как было показано ранее, для инициализации нового списка с использованием другой коллекции следует просто вызвать в отношении этой коллекции метод `toList`:

```
scala> treeSet.toList
res50: List[String] = List(blue, green, red,
yellow)
```

Или же, если нужен массив, следует вызвать метод `toArray`:

```
scala> treeSet.toArray
res51: Array[String] = Array(blue, green, red,
yellow)
```

Обратите внимание на то, что, несмотря на неотсортированность исходного списка `colors`, элементы в списке, создаваемом вызовом `toList` в отношении `TreeSet`, стоят в алфавитном порядке. Когда в отношении коллекции вызывается `toList` или `toArray`, порядок следования элементов в получающемся в результате этого списке окажется таким же, как и порядок следования элементов, создаваемый итератором, полученным в результате ссылки на элементы этой коллекции. Поскольку итератор, принадлежащий типу `TreeSet[String]`, будет выдавать строки в алфавитном порядке, эти строки в том же порядке появятся и в списке, создаваемом в результате вызова `toList` в отношении объекта `TreeSet`.

Но следует иметь в виду, что преобразование в списки или массивы, как правило, требует копирования всех элементов коллекции и поэтому для больших коллекций может выполняться довольно медленно. Но иногда это все же приходится делать из-за уже существующих API. Кроме того, многие коллекции содержат всего несколько элементов, а при этом потери в скорости незначительны.

Преобразования между изменяемыми и неизменяемыми наборами и отображениями

Иногда возникает еще одна ситуация, требующая преобразования изменяемого набора или отображения в неизменяемый аналог или наоборот. Для выполнения этих задач следует воспользоваться технологией, показанной чуть ранее для инициализации `TreeSet` с помощью элементов списка. Она предполагает создание коллекции нового типа с использованием метода `empty` и последующим добавлением новых элементов с применением либо `++`, либо `++=` в зависимости от того, что именно приемлемо для типа целевой коллекции. Преобразование неизменяемого набора `TreeSet` из предыдущего примера в изменяемый и обратно в неизменяемый выполняется следующим образом:

```
scala> import scala.collection.mutable
import scala.collection.mutable

scala> treeSet
res52:   scala.collection.immutable.TreeSet[String]
=
    TreeSet(blue, green, red, yellow)

scala> val mutaSet = mutable.Set.empty ++= treeSet
mutaSet: scala.collection.mutable.Set[String] =
    Set(red, blue, green, yellow)
```

```
scala> val immutaSet = Set.empty ++ mutaSet
immutaSet:   scala.collection.immutable.Set[String]
=
    Set(red, blue, green, yellow)
```

Аналогичной технологией можно воспользоваться для преобразований между изменяемыми и неизменяемыми отображениями:

```
scala> val muta = mutable.Map("i" -> 1, "ii" -> 2)
muta: scala.collection.mutable.Map[String,Int] =
    Map(ii -> 2,i -> 1)

scala> val immu = Map.empty ++ muta
immu: scala.collection.immutable.Map[String,Int] =
    Map(ii -> 2, i -> 1)
```

17.5. Кортежи

Согласно описанию, которое дано в шаге 9 главы 3, кортеж объединяет фиксированное количество элементов, позволяя выполнять их передачу в виде единого целого. В отличие от массива или списка, кортеж может содержать объекты различных типов. Вот как, к примеру, выглядит кортеж, содержащий целое число, строку и консоль:

```
(1, "hello", Console)
```

Кортежи избавляют вас от скуки определения упрощенных, насыщенных данными классов. Даже при том что создание класса не составляет особого труда, оно все же требует приложения конкретного количества порой напрасных усилий. Кортежи избавляют вас от необходимости выбора имени класса, области

видимости, в которой определяется класс, и имен для составляющих класса. Если класс просто хранит целое число и строку, добавление класса по имени `AnIntegerAndAString` особой ясности не внесет.

Поскольку кортежи могут сочетать объекты различных типов, они не являются наследниками класса `Traversable`. Если потребуется сгруппировать исключительно одно целое число и исключительно одну строку, то понадобится кортеж, а не `List` или `Array`.

Довольно часто кортежи применяются для возвращения из метода нескольких значений. Рассмотрим, к примеру, метод, выполняющий поиск самого длинного слова в коллекции и возвращающий наряду с ним его индекс:

```
def longestWord(words: Array[String]) = {
  var word = words(0)
  var idx = 0
  for (i <- 1 until words.length)
    if (words(i).length > word.length) {
      word = words(i)
      idx = i     }
  (word, idx)
}
```

А вот пример использования этого метода:

```
scala> val longest =
         longestWord("The quick brown fox".split(" "))
longest: (String, Int) = (quick,1)
```

Функция `longestWord` выполняет здесь два вычисления, получая при этом слово `word`, являющееся в массиве самым длинным, и индекс этого слова `idx`. Чтобы ничего не усложнять, в

функции предполагается, что список имеет хотя бы одно слово, и эта функция отдает предпочтение тому из одинаковых по длине слов, которое стоит в списке первым. Как только функция выберет, какое слово и какой индекс возвращать, она возвращает их разом, используя синтаксис кортежа `(word, idx)`.

Для доступа к элементам кортежа можно воспользоваться методом `_1`, чтобы получить первый элемент, методом `_2`, чтобы получить второй элемент, и т. д.:

```
scala> longest._1
res53: String = quick
scala> longest._2
res54: Int = 1
```

Кроме того, значение каждого элемента кортежа может быть присвоено своей собственной переменной[108]:

```
scala> val (word, idx) = longest
word: String = quick
idx: Int = 1

scala> word
res55: String = quick
```

Кстати, если не поставить круглые скобки, будет получен совершенно иной результат:

```
scala> val word, idx = longest
word: (String, Int) = (quick,1)
idx: (String, Int) = (quick,1)
```

Этот синтаксис дает несколько определений одного и того же выражения. Каждая переменная инициализируется своим собственным вычислением выражения в правой части. В данном

случае неважно, что это выражение вычисляется в кортеж. Обе переменные инициализируются всем кортежем целиком. Ряд примеров, в которых удобно применять повторные определения, можно увидеть в главе 18.

Следует заметить, что кортежи довольно просты в использовании. Они очень хорошо подходят при объединении данных, не имеющих никакого другого смысла, кроме «А и Б». Но когда объединение имеет какое-либо значение или нужно добавить к объединению какие-то методы, лучше пойти дальше и создать класс. Например, не стоит использовать кортеж из трех значений для объединения месяца, дня и года — нужно создать класс `Date`. Этим вы явно обозначите свои намерения, что сделает код понятнее для читателей и позволит компилятору и средствам самого языка помочь вам отловить возможные ошибки.

Резюме

В этой главе был дан обзор библиотеки коллекций Scala и рассмотрены наиболее важные ее классы и трейты. Опираясь на полученные знания, вы сможете эффективно работать с коллекциями Scala и будете знать, что именно нужно искать в Scaladoc, когда возникнет необходимость в дополнительных сведениях. Более подробную информацию о коллекциях Scala можно найти в главах 24 и 25. А в следующей главе внимание будет переключено с библиотеки Scala на сам язык и нам предстоит рассмотреть имеющуюся в Scala поддержку изменяемых объектов.

104 Разница в несоответствии массивов в Scala массивам в Java, то есть является ли Array[String] подтипом Array[AnyRef], будет рассмотрена в разделе 19.3.

105 Похожий пример представлен в главе 1.

106 Более подробно ключевое слово type будет рассмотрено в разделе 20.6.

107 Под одним объектом, как следует из табл. 17.3 и 17.4, понимается экземпляр одного из классов: от Set1 до Set4 или от Map1 до Map4.

108 Этот синтаксис является, по сути, особым случаем поиска по шаблону, подробно рассмотренным в разделе 15.7.

18. Изменяемые объекты

В предыдущих главах в центре внимания были функциональные (неизменяемые) объекты. Дело в том, что идея использования объектов без какого-либо изменяемого состояния заслуживала более пристального рассмотрения. Но в Scala также вполне возможно определять объекты с изменяемым состоянием. Такие изменяемые объекты зачастую появляются естественным образом, когда нужно смоделировать объекты из реального мира, которые со временем подвергаются изменениям.

В этой главе раскрывается суть изменяемых объектов и рассматриваются предлагаемые Scala синтаксические средства для их выражения. В ней также будут выполнены более широкие исследования моделирования дискретных событий, при котором используются изменяемые объекты, а также описан внутренний предметно-ориентированный язык для определения моделируемых цифровых электронных схем.

18.1. Чем обусловливается изменяемость объекта

Принципиальную разницу между чисто функциональным и изменяемым объектами можно проследить даже без изучения реализации объектов. При вызове метода или получении значения поля по указателю в отношении функционального объекта вы всегда будете получать один и тот же результат.

Например, если есть следующий список символов:

```
val cs = List('a', 'b', 'c')
```

применение кода `cs.head` всегда будет возвращать `'a'`. То же самое произойдет, даже если между местом определения `cs` и местом, где будет применено обращение `cs.head`, над списком `cs` будет проделано произвольное количество других операций.

Что же касается изменяемого объекта, то результат вызова метода или обращения к полю может зависеть от того, какие операции были ранее выполнены в отношении объекта. Хорошим примером изменяемого объекта является банковский счет. Его упрощенная реализация показана в листинге 18.1.

Листинг 18.1. Изменяемый класс банковского счета

```
class BankAccount {

  private var bal: Int = 0

  def balance: Int = bal

  def deposit(amount: Int) = {
    require(amount > 0)
    bal += amount
  }

  def withdraw(amount: Int): Boolean =
    if (amount > bal) false
    else {
      bal -= amount
      true
    }
}
```

В классе `BankAccount` определяются закрытая переменная `bal` и три открытых метода: `balance`, возвращающий текущий баланс, `deposit`, добавляющий к `bal` заданную сумму, и `withdraw`, предпринимающий попытку вывести из `bal` заданную сумму, гарантируя при этом, что баланс не станет отрицательным.

Возвращаемое withdraw значение, имеющее тип Boolean, показывает, были ли запрошенные средства успешно выведены.

Даже если ничего не знать о внутренней работе класса BankAccount, все же можно сказать, что экземпляры BankAccounts являются изменяемыми объектами:

```
scala> val account = new BankAccount
account: BankAccount = BankAccount@21cf775d

scala> account deposit 100

scala> account withdraw 80
res1: Boolean = true

scala> account withdraw 80
res2: Boolean = false
```

Заметьте, что в двух последних операциях вывода средств в ходе работы с программой были возвращены разные результаты. По итогам первой операции было возвращено значение true, так как на банковском счету содержался достаточный объем, позволяющий вывести средства. Вторая операция вывода средств была такой же, как и первая, однако в результате ее выполнения было возвращено значение false, поскольку баланс счета уменьшился настолько, что уже не мог покрыть запрошенные средства. Исходя из этого, мы понимаем, что банковским счетам присуще изменяемое состояние, так как в результате одной и той же операции в разное время получаются разные результаты.

Можно подумать, что изменяемость BankAccount априори не вызывает сомнений, поскольку в нем содержится определение var-переменной. Изменяемость и var-переменные обычно идут рука об руку, но ситуация не всегда бывает столь очевидной. Например, класс может быть изменяемым и без определения или

наследования каких-либо var-переменных, поскольку он перенаправляет вызовы методов другим объектам, которые находятся в изменяемом состоянии. Может сложиться и обратная ситуация: класс содержит var-переменные и все же он чисто функциональный. В качестве примера можно привести класс, кэширующий результаты затратной операции в поле в целях оптимизации. Чтобы подобрать пример, предположим, что имеется неоптимизированный класс Keyed с затратной операцией computeKey:

```
class Keyed {
    def computeKey: Int = ... // это займет
некоторое время
  ...
}
```

При условии, что computeKey не читает и не записывает никаких var-переменных, эффективность Keyed можно увеличить путем добавления кэша:

```
class MemoKeyed extends Keyed {
  private var keyCache: Option[Int] = None
  override def computeKey: Int = {
          if (!keyCache.isDefined) keyCache =
Some(super.computeKey)
      keyCache.get
  }
}
```

Использование MemoKeyed вместо Keyed может ускорить работу, поскольку, когда результат выполнения операции computeKey будет востребован повторно, вместо еще одного запуска computeKey может быть возвращено значение, сохраненное в поле keyCache. Но за исключением такого

ускорения поведение классов Keyed и MemoKeyed абсолютно одинаково. Следовательно, если Keyed является чисто функциональным классом, таковым будет и класс MemoKeyed, даже при том что он содержит переменную, значение которой можно присвоить заново.

18.2. Переменные и свойства с возможностью присваивания нового значения

В отношении переменной, значение которой можно присвоить заново, допускается выполнение двух основных операций: получения ее значения или присваивания ей нового значения. В таких библиотеках, как JavaBeans, эти операции часто инкапсулированы в отдельные методы считывания и записи значения, которые необходимо объявлять явно.

В Scala каждая var-переменная представляет собой незакрытый элемент какого-либо объекта, в отношении которого в нем подразумеваемо определяются методы считывания и записи значения. Но названия таких методов отличаются от предписанных соглашениями Java. Метод получения значения (get-метод) var-переменной x называется просто x, а метод присваивания значения (set-метод) — x_=.

Например, определение var-переменной, если оно появляется в классе,

```
var hour = 12
```

в добавление к полю с возможностью установки нового значения создает get-метод hour и set-метод hour_=. У поля всегда имеется пометка private[this], означающая, что значение для него может устанавливаться только из содержащего это поле объекта. В то же время get- и set-методы обеспечивают исходной var-переменной некоторую видимость. Если var-переменная

объявлена открытой (public), то таковыми же являются и методы получения и установки значения. Если она является защищенной (protected), то и они являются защищенными, и т. д.

Рассмотрим, к примеру, класс `Time`, показанный в листинге 18.2, в котором определены две открытые `var`-переменные с именами `hour` и `minute`.

Листинг 18.2. Класс с открытыми var-переменными

```
class Time {
  var hour = 12
  var minute = 0
}
```

Эта реализация в точности соответствует определению класса, показанного в листинге 18.3. В этом определении имена локальных полей `h` и `m` были выбраны произвольно, чтобы они не конфликтовали с уже используемыми именами.

Листинг 18.3. Как открытые var-переменные раскрываются в методы получения и присваивания значений

```
class Time {

  private[this] var h = 12
  private[this] var m = 0

  def hour: Int = h
  def hour_=(x: Int) = { h = x }

  def minute: Int = m
  def minute_=(x: Int) = { m = x }
```

```
}
```

Интересным аспектом такого раскрытия `var`-переменных в методы получения и присваивания значений является то, что вместо определения `var`-переменной можно также выбрать вариант непосредственного определения этих методов. Явно определяя эти методы доступа, можно как угодно интерпретировать операции доступа к переменной и присваивания ей значения. Например, вариант класса `Time`, показанный в листинге 18.4, содержит необходимые условия, благодаря которым перехватываются все присваивания недопустимых значений часам и минутам, хранящимся в переменных `hour` и `minute`.

Листинг 18.4. Непосредственное определение методов получения и присваивания значений

```
class Time {

  private[this] var h = 12
  private[this] var m = 0

  def hour: Int = h   def hour_= (x: Int) = {
    require(0 <= x && x < 24)
    h = x
  }

  def minute = m   def minute_= (x: Int) = {
    require(0 <= x && x < 60)
    m = x
  }
}
```

В некоторых языках для этих похожих на переменные величин,

которые не являются простыми переменными из-за того, что их методы получения и присваивания значений могут быть переопределены, имеются специальные синтаксические конструкции. Например, в C# эту роль выполняют свойства. По сути, принятое в Scala соглашение о постоянной интерпретации переменной как имеющей пару методов получения и присваивания значения предоставляет вам такие же возможности, что и свойства C#, но при этом не требует какого-то специального синтаксиса.

Свойства могут иметь множество назначений. В примере, показанном в листинге 18.4, методы присваивания значений навязывают соблюдение конкретных условий, защищая таким образом переменную от присваивания ей недопустимых значений. Свойствами можно воспользоваться также для регистрации всех обращений к переменной со стороны методов получения и присваивания ее значения. Или же можно объединять переменные с событиями, например путем уведомления с помощью методов-подписчиков о каждом изменении переменной (соответствующие примеры будут показаны в главе 35).

Также возможно, а иногда и полезно определять get- и set-методы без связанных с ними полей. Например, в листинге 18.5 показан класс `Thermometer`, в котором инкапсулирована переменная `temperature`, позволяющая читать и обновлять ее значение. Температурные значения могут выражаться в градусах Цельсия или Фаренгейта. Этот класс позволяет получать и устанавливать значение температуры в любых единицах измерения.

Листинг 18.5. Определение методов получения и присваивания значения без связанного с ними поля

```
class Thermometer {
```

```
  var celsius: Float = _

  def fahrenheit = celsius * 9 / 5 + 32
  def fahrenheit_= (f: Float) = {
    celsius = (f - 32) * 5 / 9
  }
   override def toString = fahrenheit + "F/" +
celsius + "C"
}
```

В первой строке тела этого класса определяется var-переменная `celsius`, в которой будет храниться значение температуры в градусах Цельсия. Для переменной `celsius` изначально устанавливается значение по умолчанию путем указания для нее в качестве инициализирующего значения знака `_`. Точнее, инициализатором поля `= _` этому полю присваивается нулевое значение. Суть нулевого значения зависит от типа поля. Для числовых типов это `0`, для булевых — `false`, а для ссылочных — `null`. Получается то же самое, что и при определении в Java некой переменной без инициализатора.

Учтите, что в Scala просто отбросить инициализатор `= _` нельзя. Если использовать код:

```
var celsius: Float
```

то получится объявление абстрактной, а не инициализированной переменной[109].

За определением переменной `celsius` следуют get-метод по имени `fahrenheit` и set-метод `fahrenheit_=`, которые обращаются к той же температуре, но в градусах Фаренгейта. В листинге нет отдельного поля, содержащего значение текущей температуры в градусах Фаренгейта. Вместо этого методы получения и установки значения для значений в градусах Фаренгейта выполняют автоматическое преобразование из

градусов Цельсия в градусы этой же температурной шкалы. Пример взаимодействия с объектом Thermometer выглядит следующим образом:

```
scala> val t = new Thermometer
t: Thermometer = 32.0F/0.0C

scala> t.celsius = 100
t.celsius: Float = 100.0

scala> t
res3: Thermometer = 212.0F/100.0C

scala> t.fahrenheit = -40
t.fahrenheit: Float = -40.0

scala> t
res4: Thermometer = -40.0F/-40.0C
```

18.3. Практический пример: моделирование дискретных событий

Далее в главе на расширенном примере будут показаны интересные способы возможного сочетания изменяемых объектов с функциями, являющимися значениями первого класса. Речь идет о конструкции и реализации симулятора цифровых схем. Эта задача разбита на несколько подзадач, каждая из которых интересна сама по себе.

Сначала будет показан весьма лаконичный язык для цифровых схем. Определение этого языка высветит основной метод встраивания предметно-ориентированных языков в язык их реализации, подобный Scala. Затем будет представлена простая, но всеобъемлющая среда для моделирования дискретных событий. Ее

основной задачей станет отслеживание действий, выполняемых в ходе моделирования. И наконец, будет показано, как могут структурироваться и создаваться программы дискретного моделирования. Целями создания таких программ являются моделирование физических объектов объектами-симуляторами и использование среды для моделирования физического времени.

Этот пример взят из классического учебного пособия Structure and Interpretation of Computer Programs[110]. Наша ситуация отличается тем, что языком реализации является Scala, а не Scheme, и тем, что различные аспекты примера структурно выделены в четыре программных уровня: первый относится к среде моделирования, второй — к основному пакету моделирования схем, третий касается библиотеки определяемых пользователем электронных схем, а четвертый предназначен для каждой моделируемой схемы как таковой. Каждый уровень выражен в виде класса, и более конкретные уровни являются наследниками более общих.

режим ускоренного чтения

Чтобы разобраться с примером моделирования дискретных событий, представленным в данной главе, потребуется некоторое время. Если вы считаете, что его лучше было бы потратить на продолжение изучения самого языка Scala, то можете перейти к чтению следующей главы.

18.4. Язык для цифровых схем

Начнем с краткого языка для описания цифровых схем, состоящих из проводников и функциональных блоков. По проводникам проходят сигналы, преобразованием которых занимаются функциональные блоки. Сигналы представлены булевыми значениями, где `true` используется для сигнала высокого уровня, а

`false` — для сигнала низкого уровня.

Основные функциональные блоки (или логические элементы) показаны на рис. 18.1.

- Блок «НЕ» выполняет над входным сигналом операцию НЕ (отрицание).
- Блок «И» устанавливает на своем выходе результат выполнения операции И (конъюнкции) над сигналами на входе.
- Блок «ИЛИ» устанавливает на своем выходе результат выполнения операции ИЛИ (дизъюнкции) над сигналами на входе.

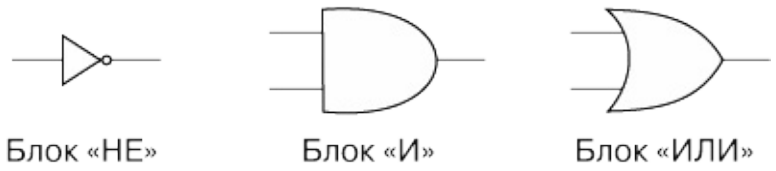


Рис. 18.1. Логические элементы

Этих логических элементов вполне достаточно для построения всех остальных функциональных блоков. У логических элементов существуют задержки, следовательно, сигнал на выходе элемента будет изменяться через некоторое время после изменения сигнала на его входе.

Элементы цифровой схемы будут описаны с применением набора классов и функций Scala. Сначала создадим класс `Wire` для проводников. Эти проводники можно сконструировать следующим образом:

```
val a = new Wire
val b = new Wire
val c = new Wire
```

или то же самое, но покороче:

```
val a, b, c = new Wire
```

Затем понадобятся три процедуры, создающие логические элементы:

```
def inverter(input: Wire, output: Wire)
def andGate(a1: Wire, a2: Wire, output: Wire)
def orGate(o1: Wire, o2: Wire, output: Wire)
```

Необычно то, что в силу имеющегося в Scala функционального уклона логические элементы в этих процедурах вместо возвращения в качестве результата сконструированных элементов конструируются в виде побочных эффектов. Например, вызов `inverter(a, b)` помещает элемент «НЕ» между проводниками `a` и `b`. Получается, что эта конструкция, основанная на побочном эффекте, позволяет упростить постепенное создание все более сложных схем. Также, при том что имена большинства методов происходят от глаголов, имена методов в Scala происходят от существительных, показывающих, какой именно элемент создается. Тем самым отображается декларативная природа DSL-языка: он должен давать описание электронной схемы, а не выполняемых в ней действий.

Из логических элементов могут создаваться более сложные функциональные блоки. Например, метод, показанный в листинге 18.6, создает полусумматор. Метод `halfAdder` получает два входных параметра, `a` и `b`, и выдает сумму `s`, определяемую как `s = (a + b) % 2`, и перенос в следующий разряд `c`, определяемый как `c = (a + b) / 2`. Схема полусумматора показана на рис. 18.2.

Листинг 18.6. Метод halfAdder

```
def halfAdder(a: Wire, b: Wire, s: Wire, c: Wire)
= {
  val d, e = new Wire
```

```
  orGate(a, b, d)
  andGate(a, b, c)
  inverter(c, e)
  andGate(d, e, s)
}
```

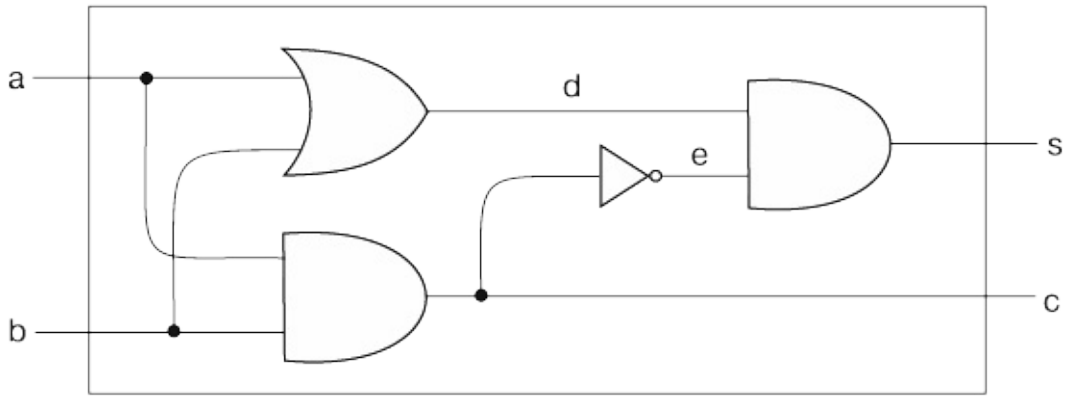


Рис. 18.2. Схема полусумматора

Обратите внимание на то, что `halfAdder` является параметризованным функциональным блоком, так же как и три метода, составляющих логические элементы. Метод `halfAdder` можно использовать для составления более сложных схем. Например, в листинге 18.7 определяется полный одноразрядный сумматор (рис. 18.3), получающий два входных параметра, `a` и `b`, а также перенос из младшего разряда (carry-in) `cin` и выдающий на выходе значение `sum`, определяемое как `sum = (a + b + cin) % 2`, и перенос в старший разряд (carry-out), определяемый как `cout = (a + b + cin) / 2`.

Листинг 18.7. Метод fullAdder

```
def fullAdder(a: Wire, b: Wire, cin: Wire,
    sum: Wire, cout: Wire) = {

  val s, c1, c2 = new Wire
```

```
    halfAdder(a, cin, s, c1)
    halfAdder(b, s, sum, c2)
    orGate(c1, c2, cout)
  }
```

Рис. 18.3. Схема сумматора

Класс `Wire` и функции `inverter`, `andGate` и `orGate` представляют собой краткий язык, с помощью которого пользователи могут определять цифровые схемы. Это неплохой пример внутреннего DSL — предметно-ориентированного языка, определенного не в виде некой самостоятельной реализации, а в виде библиотеки в языке его реализации.

Реализацию DSL-языка электронных логических схем еще предстоит разработать. Поскольку целью определения схемы средствами DSL является моделирование электронной схемы, вполне разумно будет основой реализации DSL сделать общий API для моделирования дискретных событий. В следующих двух разделах будет представлен первый API моделирования, а затем в качестве надстройки над ним — реализация DSL-языка электронных логических схем.

18.5. API моделирования

API моделирования показан в листинге 18.8. Он состоит из класса `Simulation` в `packageorg.stairwaybook.simulation`. Наследниками этого класса являются определенные библиотеки моделирования, наращивающие его предметно-ориентированную функциональность. В этом разделе представлены элементы класса

Simulation.

Листинг 18.8. Класс Simulation

```
abstract class Simulation {

  type Action = () => Unit

  case class WorkItem(time: Int, action: Action)

  private var curtime = 0
  def currentTime: Int = curtime

  private var agenda: List[WorkItem] = List()

  private def insert(ag: List[WorkItem],
      item: WorkItem): List[WorkItem] = {

     if (ag.isEmpty || item.time < ag.head.time)
item :: ag
    else ag.head :: insert(ag.tail, item)
  }

  def afterDelay(delay: Int)(block: => Unit) = {
    val item = WorkItem(currentTime + delay, () =>
block)
    agenda = insert(agenda, item)
  }

  private def next() = {
    (agenda: @unchecked) match {
      case item :: rest =>
```

```
        agenda = rest
        curtime = item.time
        item.action()
      }
    }
    def run() = {
      afterDelay(0) {
        println("*** simulation started, time = " +
            currentTime + " ***")
      }
      while (!agenda.isEmpty) next()
    }
  }
```

При моделировании дискретных событий действия, определенные пользователем, выполняются в указанные моменты времени. Действия, определенные конкретными подклассами моделирования, совместно используют один и тот же тип:

```
type Action = () => Unit
```

Эта инструкция определяет `Action` в качестве псевдонима типа процедуры, получающей пустой список параметров и возвращающей тип `Unit`. Тип `Action` является элементом типа класса `Simulation`. Его можно рассматривать как гораздо легче читаемое имя для типа `() => Unit`. Элементы типов будут подробно рассмотрены в разделе 20.6.

Момент времени, в который выполняется действие, является моментом моделирования — он не имеет ничего общего со временем «настенных часов». Моменты времени моделирования представлены просто как целые числа. Текущий момент времени моделирования хранится в закрытой переменной:

```
private var curtime: Int = 0
```

У переменной имеется открытый метод доступа, извлекающий текущее время:

```
def currentTime: Int = curtime
```

Это сочетание закрытой переменной с открытым методом доступа используется, чтобы гарантировать невозможность изменения текущего времени за пределами класса `Simulation`. Обычно не нужно, чтобы моделируемые вами объекты манипулировали текущим временем, за исключением, возможно, случая, когда моделируется путешествие во времени.

Действие, которое должно быть выполнено в указанное время, называется рабочим элементом. Рабочие элементы реализуются следующим классом:

```
case class WorkItem(time: Int, action: Action)
```

Класс `WorkItem` сделан case-классом, чтобы иметь возможность получить следующие синтаксические удобства: для создания экземпляров класса можно использовать фабричный метод `WorkItem` и при этом без каких-либо усилий получить средства доступа к параметрам конструктора `time` и `action`. Следует также заметить, что класс `WorkItem` вложен в класс `Simulation`. Вложенные классы в Scala рассматриваются аналогично таким же классам в Java. Более подробно этот вопрос мы изучим в разделе 20.7.

В классе `Simulation` хранится *план действий* (agenda) всех остальных рабочих элементов, которые еще не были выполнены. Рабочие элементы отсортированы по моделируемому времени, в которое они должны быть запущены:

```
private var agenda: List[WorkItem] = List()
```

Перечень действий будет сохраняться в надлежащем отсортированном порядке благодаря использованию метода

insert, который обновляет этот перечень. Вызов метода insert в качестве единственного способа добавления рабочего элемента к перечню действий можно увидеть из метода afterDelay:

```
def afterDelay(delay: Int)(block: => Unit) = {
   val item = WorkItem(currentTime + delay, () =>
block)
  agenda = insert(agenda, item)
}
```

Как следует из названия, этот метод вставляет действие, задаваемое блоком, в план действий, планируя время задержки его выполнения delay после текущего момента моделируемого времени. Например, следующий вызов создаст выполнение нового рабочего элемента в моделируемое время currentTime + delay:

```
afterDelay(delay) { count += 1 }
```

Код, предназначенный для выполнения, содержится во втором аргументе метода. Формальный параметр имеет тип => Unit, то есть это вычисление типа Unit, передаваемое по имени. Следует напомнить, что параметры до востребования (by-name parameters) при передаче методу не вычисляются. Следовательно, в показанном ранее вызове значение count будет увеличено на единицу, только когда среда моделирования вызовет действие, сохраненное в рабочем элементе. Обратите внимание на то, что afterDelay является каррированной функцией. Это хороший пример разъясненного в разделе 9.5 принципа, согласно которому каррирование может использоваться для выполнения вызовов тех методов, которые больше похожи на встроенный синтаксис языка.

Созданный рабочий элемент еще нужно вставить в план действий. Это делается посредством метода insert, в котором поддерживается предварительное условие отсортированности плана действий по времени:

```
private def insert(ag: List[WorkItem],
    item: WorkItem): List[WorkItem] = {

  if (ag.isEmpty || item.time < ag.head.time) item
:: ag
  else ag.head :: insert(ag.tail, item)
}
```

Ядро класса `Simulation` определяется методом `run`:

```
def run() = {
  afterDelay(0) {
    println("*** simulation started, time = " +
        currentTime + " ***")
  }
  while (!agenda.isEmpty) next()
}
```

Этот метод периодически берет первый элемент из плана действий, удаляет его из этого плана и выполняет данный рабочий элемент. Он продолжает свою работу до тех пор, пока в плане действий не останется элементов для выполнения. Каждый шаг делается вызовом метода `next`, имеющего следующее определение:

```
private def next() = {
  (agenda: @unchecked) match {
    case item :: rest =>
      agenda = rest
      curtime = item.time
      item.action()
  }
}
```

В методе `next` текущий план действий разбивается с помощью

поиска по шаблону на передовой элемент item и весь остальной перечень рабочих элементов rest. Передовой элемент из текущего плана действий удаляется, моделируемое время curtime устанавливается на время рабочего элемента, и выполняется действие рабочего элемента.

Обратите внимание на то, что next может быть вызван, только если план действий еще не пуст. Варианта для пустого перечня нет, поэтому при попытке запуска next в отношении пустого плана действий будет выдано исключение MatchError.

В действительности же компилятор Scala обычно выдает предупреждение, что для списка не указан один из возможных шаблонов:

```
Simulator.scala:19: warning: match is not exhaustive!
missing combination Nil

    agenda match {
    ^
one warning found
```

В данном случае неуказанный вариант никакой проблемы не создает, поскольку известно, что next вызывается только в отношении непустого плана действий. Поэтому может возникнуть желание отключить предупреждение. Как было показано в разделе 15.5, это может быть сделано добавлением к выражению селектора поиска по шаблону аннотации @unchecked. Именно поэтому в коде Simulation используется (agenda: @unchecked) match, а не agenda match. И это правильно. Объем кода для среды моделирования может показаться весьма скромным. Возникает вопрос: а как эта среда вообще может поддерживать

содержательное моделирование, если она лишь выполняет перечень рабочих элементов? В действительности же эффективность среды моделирования определяется тем фактом, что действия, сохраненные в рабочих элементах, в ходе своего выполнения могут самостоятельно устанавливать дальнейшие рабочие элементы в план действий. Тем самым открывается возможность получения из вычисления простых начальных действий довольно продолжительных моделирующих действий.

18.6. Моделирование электронной логической схемы

Следующим шагом станет использование среды моделирования с целью реализации предметно-ориентированного языка для логических схем, показанного в разделе 18.4. Следует напомнить, что DSL логических схем состоит из класса для проводников и методов, создающих логические элементы И, ИЛИ и НЕ. Все это содержится в классе `BasicCircuitSimulation`, который расширяет среду моделирования. Этот класс показан в листингах 18.9 и 18.10.

Листинг 18.9. Первая половина класса BasicCircuitSimulation

```
package org.stairwaybook.simulation

abstract class BasicCircuitSimulation extends
Simulation {

  def InverterDelay: Int
  def AndGateDelay: Int
  def OrGateDelay: Int

  class Wire {
```

```
  private var sigVal = false
  private var actions: List[Action] = List()

  def getSignal = sigVal

  def setSignal(s: Boolean) =
    if (s != sigVal) {
      sigVal = s
      actions foreach (_ ())
    }

  def addAction(a: Action) = {
    actions = a :: actions
    a()
  }
}

def inverter(input: Wire, output: Wire) = {
  def invertAction() = {
    val inputSig = input.getSignal
    afterDelay(InverterDelay) {
      output setSignal !inputSig
    }
  }
  input addAction invertAction
}

// продолжение см. в листинге 18.10...
```

Листинг 18.10. Вторая половина класса BasicCircuitSimulation

```
// ...начало см. в листинге 18.9
```

```
  def andGate(a1: Wire, a2: Wire, output: Wire) =
{
    def andAction() = {
      val a1Sig = a1.getSignal
      val a2Sig = a2.getSignal
      afterDelay(AndGateDelay) {
        output setSignal (a1Sig & a2Sig)
      }
    }
    a1 addAction andAction
    a2 addAction andAction
  }

  def orGate(o1: Wire, o2: Wire, output: Wire) = {
    def orAction() = {
      val o1Sig = o1.getSignal
      val o2Sig = o2.getSignal
      afterDelay(OrGateDelay) {
        output setSignal (o1Sig | o2Sig)
      }
    }
    o1 addAction orAction
    o2 addAction orAction
  }

  def probe(name: String, wire: Wire) = {
    def probeAction() = {
      println(name + " " + currentTime +
          " new-value = " + wire.getSignal)
    }
    wire addAction probeAction
  }
```

```
}
```

В классе `BasicCircuitSimulation` объявляются три абстрактных метода, представляющих задержки основных логических элементов: `InverterDelay`, `AndGateDelay` и `OrGateDelay`. Настоящие задержки на уровне этого класса неизвестны, поскольку они зависят от технологии моделируемых логических микросхем. Поэтому задержки в классе `BasicCircuitSimulation` остаются абстрактными и их конкретное определение делегируется подклассам[111]. Далее будет рассмотрена реализация остальных элементов класса `BasicCircuitSimulation`.

Класс Wire

Проводникам нужно поддерживать три основных действия:

- `getSignal: Boolean` возвращает текущий сигнал в проводник;
- `setSignal(sig: Boolean)` выставляет сигнал проводника в `sig`;
- `addAction(p: Action)` прикрепляет указанную процедуру `p` к действиям проводника. Замысел заключается в том, чтобы все процедуры действий, прикрепленные к какому-либо проводнику, выполнялись всякий раз, когда сигнал на проводнике изменяется. Как правило, действия добавляются к проводнику подключенными к нему компонентами. Прикрепленное действие выполняется в момент его добавления к проводнику, а после этого всякий раз при изменении сигнала на проводнике.

Реализация класса `Wire` имеет следующий вид:

```
class Wire {

  private var sigVal = false
  private var actions: List[Action] = List()

  def getSignal = sigVal

  def setSignal(s: Boolean) =
    if (s != sigVal) {
      sigVal = s
      actions foreach (_ ())
    }

  def addAction(a: Action) = {
    actions = a :: actions
    a()
  }
}
```

Состояние проводника формируется двумя закрытыми переменными. Переменная `sigVal` представляет текущий сигнал, а переменная `actions` — процедуры действий, прикрепленные в данный момент к проводнику. В реализациях методов представляет интерес только та часть, которая относится к методу `setSignal`: когда сигнал проводника изменяется, в переменной `sigVal` сохраняется новое значение. Кроме того, выполняются все действия, прикрепленные к проводнику. Обратите внимание на используемую для этого сокращенную форму синтаксиса: выражение `actions foreach (_ ())` вызывает применение функции `_ ()` к каждому элементу в перечне действий. В соответствии с описанием, приведенным в разделе 8.5, функция `_ ()` является сокращенной формой записи для `f => f ()`, то есть она получает функцию (назовем ее `f`) и применяет ее к пустому

списку параметров.

Метод inverter

Единственным результатом создания метода `inverter` является то, что действие устанавливается на его входящем проводнике. Это действие вызывается при его установке, а затем всякий раз при изменении сигнала на входе. Эффект от действия заключается в установке выходного значения (посредством `setSignal`) на отрицание его входного значения. Поскольку у логического элемента НЕ имеется задержка, это изменение должно наступить только по прошествии определенного количества единиц моделируемого времени, хранящегося в переменной `InverterDelay`, после того как изменилось входное значение и было выполнено действие. Эти обстоятельства подсказывают следующий вариант реализации:

```
def inverter(input: Wire, output: Wire) = {
  def invertAction() = {
    val inputSig = input.getSignal
    afterDelay(InverterDelay) {
      output setSignal !inputSig
    }
  }
  input addAction invertAction
}
```

Эффект метода `inverter` заключается в добавлении действия `invertAction` к входящему проводнику. При вызове этого действия берется входной сигнал и устанавливается еще одно действие, инвертирующее выходной сигнал в плане действий моделирования. Это другое действие должно быть выполнено по прошествии того количества единиц времени моделирования, которое хранится в переменной `InverterDelay`. Обратите

внимание на то, как для создания нового рабочего элемента, который предназначен для выполнения в будущем, в методе используется метод `afterDelay`.

Методы andGate и orGate

Реализация моделирования логического элемента И аналогична реализации моделирования элемента НЕ. Целью является выставление на выходе конъюнкции его входных сигналов. Это должно произойти по прошествии того количества единиц времени моделирования, которое хранится в переменной `AndGateDelay`, после того как изменится любой из его двух входных сигналов. Стало быть, подойдет следующая реализация:

```
def andGate(a1: Wire, a2: Wire, output: Wire) = {
  def andAction() = {
    val a1Sig = a1.getSignal
    val a2Sig = a2.getSignal
    afterDelay(AndGateDelay) {
      output setSignal (a1Sig & a2Sig)
    }
  }
  a1 addAction andAction
  a2 addAction andAction
}
```

Эффект от вызова метода `andGate` заключается в добавлении действия `andAction` к обоим входным проводникам, `a1` и `a2`. При вызове этого действия берутся оба входных сигнала и устанавливается еще одно действие, которое выдает выходной сигнал в виде конъюнкции обоих входных сигналов. Это другое действие должно быть выполнено по прошествии того количества единиц времени моделирования, которое хранится в переменной `AndGateDelay`. Учтите, что при смене любого сигнала на входящих

проводниках выход должен вычисляться заново. Именно поэтому одно и то же действие andAction устанавливается на каждом из двух входящих проводников, a1 и a2. Метод orGate реализуется аналогичным образом, за исключением того, что он моделирует логическую операцию ИЛИ, а не логическую операцию И.

Вывод, получаемый в результате моделирования

Для запуска симулятора нужен способ проверки изменений сигналов на проводниках. Для выполнения этой задачи можно смоделировать действие по помещению пробника на проводник:

```
def probe(name: String, wire: Wire) = {
  def probeAction() = {
    println(name + " " + currentTime +
        " new-value = " + wire.getSignal)
  }
  wire addAction probeAction
}
```

Эффект от процедуры probe заключается в установке на заданный проводник действия probeAction. Как обычно, такое действие выполняется всякий раз при изменении сигнала на проводнике. В данном случае он просто выводит на стандартное устройство название проводника (которое передается probe в качестве первого параметра), а также текущее моделируемое время и новое значение проводника.

Запуск симулятора

После всех приготовлений настало время посмотреть на симулятор в действии. Для определения конкретного моделирования нужно оформить наследование из класса среды моделирования. Чтобы увидеть кое-что интересное, будет создан абстрактный класс

моделирования, расширяющий BasicCircuitSimulation и содержащий определения методов для полусумматора и сумматора в том виде, в котором они были представлены в листингах 18.6 и 18.7 соответственно. Этот класс, который будет назван callCircuitSimulation, показан в листинге 18.11.

Листинг 18.11. Класс CircuitSimulation

```
package org.stairwaybook.simulation

abstract class CircuitSimulation
  extends BasicCircuitSimulation {

   def halfAdder(a: Wire, b: Wire, s: Wire, c:
Wire) = {
    val d, e = new Wire
    orGate(a, b, d)
    andGate(a, b, c)
    inverter(c, e)
    andGate(d, e, s)
  }

  def fullAdder(a: Wire, b: Wire, cin: Wire,
      sum: Wire, cout: Wire) = {

    val s, c1, c2 = new Wire
    halfAdder(a, cin, s, c1)
    halfAdder(b, s, sum, c2)
    orGate(c1, c2, cout)
  }
}
```

Конкретная модель логической схемы будет объектом с наследованием из класса `CircuitSimulation`. Этому объекту по-прежнему необходимо зафиксировать задержки на логических элементах в соответствии с технологией реализации моделируемой логической микросхемы. И наконец, понадобится также определить конкретную моделируемую схему.

Эти шаги можно проделать в интерактивном режиме в интерпретаторе Scala:

```
scala> import org.stairwaybook.simulation._
import org.stairwaybook.simulation._
```

Сначала займемся задержками логических элементов. Определим объект (назовем его `MySimulation`), предоставляющий ряд чисел:

```
scala>      object      MySimulation      extends
CircuitSimulation {
         def InverterDelay = 1
         def AndGateDelay = 3
         def OrGateDelay = 5
      }
defined module MySimulation
```

Поскольку предполагается периодически получать доступ к элементам объекта `MySimulation`, импортирование этого объекта укоротит последующий код:

```
scala> import MySimulation._
import MySimulation._
```

Далее займемся схемой. Определим четыре проводника и поместим пробники на два из них:

```
scala> val input1, input2, sum, carry = new Wire
```

```
input1: MySimulation.Wire =
    BasicCircuitSimulation$Wire@111089b
input2: MySimulation.Wire =
    BasicCircuitSimulation$Wire@14c352e
sum: MySimulation.Wire =
    BasicCircuitSimulation$Wire@37a04c
carry: MySimulation.Wire =
    BasicCircuitSimulation$Wire@1fd10fa
scala> probe("sum", sum)
sum 0 new-value = false

scala> probe("carry", carry)
carry 0 new-value = false
```

Обратите внимание на то, что пробники немедленно выводят выход. Дело в том, что каждое действие, установленное на проводнике, первый раз выполняется при его установке.

Теперь определим подключение к проводникам полусумматора:

```
scala> halfAdder(input1, input2, sum, carry)
```

И наконец, установим один за другим сигналы на двух входящих проводниках на `true` и запустим моделирование:

```
scala> input1 setSignal true

scala> run()
*** simulation started, time = 0 ***
sum 8 new-value = true

scala> input2 setSignal true

scala> run()
```

```
*** simulation started, time = 8 ***
carry 11 new-value = true
sum 15 new-value = false
```

Резюме

В данной главе были собраны воедино две на первый взгляд несопоставимые технологии: изменяемое состояние и функции высшего порядка. Изменяемое состояние было использовано для моделирования физических объектов, состояние которых со временем изменяется. Функции высшего порядка были задействованы в среде моделирования для выполнения действий в указанные моменты моделируемого времени. Они также применялись в моделировании логических схем в качестве пусковых механизмов, связывающих действия с изменениями состояния. Попутно был показан простой способ определения предметно-ориентированного языка в виде библиотеки. Наверное, для одной главы этого вполне достаточно.

Если вас привлекла эта тема, можете попробовать создать дополнительные примеры моделирования. Можно объединить полусумматоры и сумматоры для формирования более крупных схем или на основе ранее определенных логических элементов разработать новые схемы и смоделировать их. В следующей главе будет рассмотрена имеющаяся в Scala параметризация типов и показан еще один пример, сочетающий в себе функциональные и императивные подходы, которые выдают весьма неплохое решение.

109 Абстрактные переменные будут рассматриваться в главе 20.

110 Abelson H., Sussman G. J. Structure and Interpretation of Computer Programs. 2nd ed. — The MIT Press, 1996.

111 Имена этих методов задержки начинаются с прописных букв, потому что они представляют собой константы. Но это методы, и они могут быть переопределены в подклассах. Как те же вопросы решаются с использованием val-переменных, будет показано в разделе 20.3.

19. Параметризация типов

В этой главе будут рассмотрены подробности имеющейся в Scala параметризации типов. Попутно мы расскажем о некоторых технологических приемах для скрытия информации, представленной в главе 13, путем использования конкретного примера — конструкции класса для чисто функциональных очередей. Параметризация типов и скрытие информации рассматриваются вместе, поскольку скрытие информации может пригодиться для получения более общих вариантов аннотаций параметризации типов.

Параметризация типов позволяет создавать обобщенные классы и трейты. Например, наборы имеют обобщенный характер и получают параметр типа: они определяются как `Set[T]`. В результате любой отдельно взятый экземпляр набора может иметь тип `Set[String]`, `Set[Int]` и т. д., но он должен быть набором чего-либо. В отличие от языка Java, в котором разрешено использование несформированных типов (raw types), Scala требует указывать параметры типа. Вариантность определяет взаимоотношения наследования параметризированных типов, к примеру таких, при которых `Set[String]` является подтипом `Set[AnyRef]`.

Глава состоит из трех частей. В первой части разрабатывается структура данных для чисто функциональных очередей. Во второй — технологические приемы скрытия внутреннего представления подробностей этой структуры. В заключительной части объясняется вариантность параметров типов и порядка их взаимодействия со скрытием информации.

19.1. Функциональные очереди

Функциональная очередь представляет собой структуру данных с тремя операциями:

- `head` — возвращает первый элемент очереди;
- `tail` — возвращает очередь без первого элемента;
- `enqueue` — возвращает новую очередь с заданным элементом, добавленным в ее конец.

В отличие от изменяемой очереди, функциональная очередь не изменяет своего содержимого при добавлении элемента. Вместо этого возвращается новая очередь, содержащая элемент. В этой главе мы нацелимся на создание класса по имени `Queue`, работающего следующим образом:

```
scala> val q = Queue(1, 2, 3)
q: Queue[Int] = Queue(1, 2, 3)

scala> val q1 = q enqueue 4
q1: Queue[Int] = Queue(1, 2, 3, 4)

scala> q
res0: Queue[Int] = Queue(1, 2, 3)
```

Если бы у `Queue` была изменяемая реализация, операция `enqueue` в показанной ранее второй строке ввода повлияла бы на содержимое `q`: по сути, после этой операции оба результата, и `q1`, и исходная очередь `q`, будут содержать последовательность `1, 2, 3, 4`. А для функциональной очереди добавленное значение обнаруживается только в результате `q1`, но не в очереди `q`, в отношении которой выполнялась операция.

Чистые функциональные очереди также немного схожи со списками. Обе эти коллекции имеют так называемую абсолютно неизменяемую структуру данных, где старые версии остаются доступными даже после расширений или модификаций. Обе они

поддерживают операции head и tail. Но список расширяется, как правило, спереди с использованием операции ::, а очередь — сзади с помощью операции enqueue.

Как добиться эффективной реализации очереди? В идеале функциональная (неизменяемая) очередь не должна иметь существенно более высоких издержек по сравнению с императивной (изменяемой) очередью. То есть все три операции, head, tail и enqueue, должны выполняться за постоянное время.

Одним из простых подходов к реализации функциональной очереди станет использование списка в качестве типа представления. Тогда head и tail будут просто преобразовываться в те же операции над списком, а enqueue станет объединением.

В таком варианте получится следующая реализация:

```
class SlowAppendQueue[T](elems: List[T]) { // Неэффективное решение
  def head = elems.head
  def tail = new SlowAppendQueue(elems.tail)
   def enqueue(x: T) = new SlowAppendQueue(elems ::: List(x))
}
```

Проблемной в данной реализации является операция enqueue. На нее уходит время, пропорциональное количеству элементов, хранящихся в очереди. Если требуется постоянное время добавления, можно также попробовать изменить в списке, представляющем очередь, порядок следования элементов на обратный, чтобы последний добавляемый элемент стал в списке первым. Тогда получится следующая реализация:

```
class SlowHeadQueue[T](smele: List[T]) { // Неэффективное решение
  // В smele обратный порядок следования элементов
```

```
  def head = smele.last
  def tail = new SlowHeadQueue(smele.init)
    def enqueue(x: T) = new SlowHeadQueue(x ::
smele)
}
```

Теперь у операции `enqueue` постоянное время выполнения, а вот у `head` и `tail` — нет. На их работу теперь уходит время, пропорциональное количеству элементов, хранящихся в очереди.

При изучении этих двух примеров реализация, в которой на все три операции будет затрачиваться постоянное время, не представляется такой уж простой. И действительно, возникают серьезные сомнения в возможности подобной реализации! Но воспользовавшись сочетанием двух операций, можно подойти к желаемому результату очень близко. Замысел состоит в представлении очереди в виде двух списков с названиями `leading` и `trailing`. Список `leading` содержит элементы, которые располагаются от конца к началу, а элементы списка `trailing` следуют из начала в конец очереди, то есть в обратном порядке. Содержимое всей очереди в любой момент времени эквивалентно коду `leading ::: trailing.reverse`.

Теперь, чтобы добавить элемент, следует просто провести конс-операцию в отношении списка `trailing`, воспользовавшись оператором `::`, и тогда операция `enqueue` будет выполняться за постоянное время. Это означает, что, если изначально пустая очередь выстраивается на основе последовательно проведенных операций `enqueue`, список `trailing` будет расти, а список `leading` останется пустым. Затем перед выполнением первой операции `head` или `tail` в отношении пустого списка `leading` весь список `trailing` копируется в `leading` в обратном порядке следования элементов. Это делается с помощью операции по имени `mirror`. Реализация очередей с использованием данного подхода показана в листинге 19.1.

Листинг 19.1. Базовая функциональная очередь

```
class Queue[T](
  private val leading: List[T],
  private val trailing: List[T]
) {
  private def mirror =
    if (leading.isEmpty)
      new Queue(trailing.reverse, Nil)
    else
      this

  def head = mirror.leading.head

  def tail = {
    val q = mirror
    new Queue(q.leading.tail, q.trailing)
  }

  def enqueue(x: T) =
    new Queue(leading, x :: trailing)
}
```

Какова вычислительная сложность этой реализации очередей? Операция `mirror` может занять время, пропорциональное количеству элементов очереди, но только если список `leading` пуст. Если список `leading` не пуст, возвращение из метода происходит немедленно. Поскольку `head` и `tail` вызывают `mirror`, их вычислительная сложность также может иметь линейную зависимость от размера очереди. Но чем длиннее становится очередь, тем реже вызывается `mirror`.

И действительно, допустим, есть очередь длиной *n* с пустым списком `leading`. Тогда операции `mirror` придется скопировать в

обратном порядке список длиной *n*. Но в следующий раз, как только список `leading` опустеет, что произойдет после *n* операций `tail`, операции `mirror` придется потрудиться. Это означает, что вы можете расплатиться за каждую из этих *n* операций `tail` одной *n*-ной вычислительной сложностью операции `mirror`, что подразумевает постоянный объем работы. При условии, что операции `head`, `tail` и `enqueue` используются примерно с одинаковой частотой, *демпфированная* вычислительная сложность является, таким образом, постоянной для каждой операции. Следовательно, функциональные очереди асимптотически обладают эффективностью, сопоставимой с эффективностью изменяемых очередей.

Но этот аргумент следует сопроводить некоторыми оговорками. Во-первых, речь шла только об асимптотическом поведении, а постоянно действующие факторы могут несколько отличаться от него. Во-вторых, аргумент основывается на том, что операции `head`, `tail` и `enqueue` вызываются с примерно одинаковой частотой. Если операция `head` вызывается намного чаще, чем остальные две, данный аргумент утратит силу, поскольку каждый вызов `head` может повлечь за собой весьма затратную реорганизацию списка с помощью операции `mirror`. Негативных последствий, названных во второй оговорке, можно избежать, поскольку есть возможность сконструировать функциональные очереди таким образом, что в череде последовательных операций `head` реорганизация потребуется только для первой из них. Как это делается, будет показано в конце главы.

19.2. Скрытие информации

Теперь по эффективности реализация класса `Queue`, показанная в листинге 19.1, нас вполне устраивает. Конечно, вы можете возразить, что за эту эффективность пришлось расплачиваться неоправданно подробной детализацией. Глобально доступный

конструктор Queue получает в качестве параметров два списка, в одном из которых элементы следуют в обратном порядке, что вряд ли совпадает с интуитивным представлением об очереди. Нам нужен способ, позволяющий скрыть данный конструктор от кода клиента. Некоторые способы решения этой задачи на Scala будут показаны в текущем разделе.

Закрытые конструкторы и фабричные методы

В Java конструктор можно скрыть, объявив его закрытым. В Scala первичный конструктор не имеет явно указываемого определения — подразумевается, что он автоматически определяется с параметрами и телом класса. Тем не менее, как показано в листинге 19.2, скрыть первичный конструктор можно, добавив перед списком параметров класса модификатор private.

Листинг 19.2. Скрытие первичного конструктора путем превращения его в закрытый

```
class Queue[T] private (
  private val leading: List[T],
  private val trailing: List[T]
)
```

Модификатор private, указанный между именем класса и его параметрами, служит признаком того, что конструктор класса Queue является закрытым: доступ к нему можно получить только изнутри самого класса и из его объекта-спутника. Имя класса Queue по-прежнему остается открытым, поэтому вы можете использовать его в качестве типа, но не можете вызвать его конструктор:

```
scala> new Queue(List(1, 2), List(3))
<console>:9: error: constructor Queue in class
```

```
Queue cannot
be accessed in object $iw
               new Queue(List(1, 2), List(3))
               ^
```

Теперь, когда первичный конструктор класса Queue уже не может быть вызван из кода клиента, нужен какой-то другой способ создания новых очередей. Одной из возможностей является добавление дополнительного конструктора:

```
def this() = this(Nil, Nil)
```

Этот конструктор, показанный в предыдущем примере, создает пустую очередь. В качестве уточнения он может получать список исходных элементов очереди:

```
def this(elems: T*) = this(elems.toList, Nil)
```

Следует напомнить, что в соответствии с описанием из раздела 8.8 T* является формой записи для повторяющихся параметров.

Еще одна возможность заключается в добавлении фабричного метода, создающего очередь из последовательности исходных элементов. Прямой путь к этому состоит в определении объекта Queue, имеющего такое же имя, как и определяемый класс, и, как показано в листинге 19.3, содержащего метод apply.

Листинг 19.3. Фабричный метод apply в объекте-спутнике

```
object Queue {
  // создает очередь с исходными элементами xs
   def apply[T](xs: T*) = new Queue[T](xs.toList,
Nil)
}
```

Помещая этот объект в тот же самый исходный файл, в котором

находится определение класса Queue, вы превращаете объект в объект-спутник данного класса. В разделе 13.5 было показано, что объект-спутник имеет такие же права доступа, что и его класс. Поэтому метод apply в объекте Queue может создать новый объект Queue, несмотря на то что конструктор класса Queue является закрытым.

Следует заметить, что по причине вызова фабричным методом метода apply клиенты могут создавать очереди с помощью выражений вида Queue(1, 2, 3). Это выражение раскрывается в Queue.apply(1, 2, 3), поскольку Queue является объектом, заменяющим функцию. В результате этого клиенты видят Queue в качестве глобально определенного фабричного метода. На самом же деле в Scala нет методов с глобальной областью видимости, поскольку каждый метод должен быть заключен в объект или класс. Но, используя методы по имени apply внутри глобальных объектов, вы можете поддерживать схемы, похожие на вызов глобальных методов.

Альтернативный вариант: закрытые классы

Закрытые конструкторы и закрытые элементы класса являются всего лишь одним из способов скрытия инициализации и представления класса. Еще одним более радикальным способом является скрытие самого класса и экспорт только трейта, показывающего публичный интерфейс класса. Код реализации такой конструкции представлен в листинге 19.4. В нем показан трейт Queue, в котором объявляются методы head, tail и enqueue. Все три метода реализованы в подклассе QueueImpl, который является закрытым подклассом внутри класса объекта Queue. Тем самым клиентам открывается доступ к той же информации, что и раньше, но с использованием другого приема. Вместо скрытия отдельно взятых конструкторов и методов в этой версии скрывается весь реализующий очереди класс.

Листинг 19.4. Абстракция типа для функциональных очередей

```
trait Queue[T] {
  def head: T
  def tail: Queue[T]
  def enqueue(x: T): Queue[T]
}

object Queue {

  def apply[T](xs: T*): Queue[T] =
    new QueueImpl[T](xs.toList, Nil)

  private class QueueImpl[T](
    private val leading: List[T],
    private val trailing: List[T]
  ) extends Queue[T] {

    def mirror =
      if (leading.isEmpty)
        new QueueImpl(trailing.reverse, Nil)
      else
        this

    def head: T = mirror.leading.head
    def tail: QueueImpl[T] = {
      val q = mirror
      new QueueImpl(q.leading.tail, q.trailing)
    }

    def enqueue(x: T) =
      new QueueImpl(leading, x :: trailing)
```

```
    }
  }
```

19.3. Аннотация вариантности

Согласно определению в листинге 19.4 Queue является трейтом, но не типом. Queue не является типом из-за получения параметра типа. В результате утрачивается возможность создания переменных типа Queue:

```
scala> def doesNotCompile(q: Queue) = {}
<console>:8: error: class Queue takes type parameters
       def doesNotCompile(q: Queue) = {}
                                  ^
```

Вместо этого Queue позволяет указывать параметризированные типы, такие как Queue[String], Queue[Int] или Queue[AnyRef]:

```
scala> def doesCompile(q: Queue[AnyRef]) = {}
doesCompile: (q: Queue[AnyRef])Unit
```

Таким образом, Queue является трейтом, а Queue[String] — типом. Queue также называют *конструктором типа*, поскольку у вас есть возможность сконструировать тип с его участием, указав параметр типа. (Это аналогично конструированию экземпляра объекта с использованием самого обычного конструктора с указанием параметра значения.) Конструктор типа Queue генерирует семейство типов, включающее Queue[Int], Queue[String] и Queue[AnyRef].

Можно также сказать, что Queue — обобщенный трейт. (Классы и трейты, получающие параметры типа, являются обобщенными, а вот типы, генерируемые ими, являются параметризированными, а

не обобщенными.) Понятие «обобщенный» означает, что вы определяете множество конкретных типов с помощью одного обобщенно написанного класса или трейта. Например, трейт `Queue` в листинге 19.4 определяет обобщенную очередь. Конкретными очередями будут `Queue[Int]`, `Queue[String]` и т. д.

Сочетание параметров типа и системы подтипов вызывает ряд интересных вопросов. Например, существуют ли какие-то особые подтиповые родственные взаимоотношения между представителями семейства типов, генерируемого `Queue[T]`? Конкретнее говоря, следует ли рассматривать `Queue[String]` в качестве подтипа `Queue[AnyRef]`? Или в более широком смысле, если `S` является подтипом `T`, следует ли рассматривать `Queue[S]` в качестве подтипа `Queue[T]`? Если да, то можно будет сказать, что трейт `Queue` допускает *ковариантность* (или гибкость) в своем параметре типа `T`. Или же, поскольку у него всего один параметр типа, можно просто сказать, что `Queue`-очереди допускают ковариантность. Такая ковариантность `Queue`-очередей будет означать, к примеру, что вы можете передать `Queue[String]` ранее показанному методу `doesCompile`, который получает параметр значения типа `Queue[AnyRef]`.

Интуитивно все это воспринимается как вполне возможный вариант, поскольку очередь из `String`-элементов похожа на частный случай очереди из `AnyRef`-элементов. Но в Scala у обобщенных типов изначально имеется *невариантное* (или жесткое) подтипирование. То есть при использовании трейта `Queue`, определенного в листинге 19.4, очереди с различными типами элементов никогда не будут состоять в подтиповых взаимоотношениях. `Queue[String]` не станет использоваться как `Queue[AnyRef]`. Но получить ковариантность (гибкость) подтипирования очередей можно следующим изменением первой строчки определения `Queue`:

```
trait Queue[+T] { … }
```

Указание знака «плюс» (+) в качестве префикса формального параметра типа показывает, что подтипирование в данном параметре является ковариантным (гибким). Добавляя этот единственный знак, вы сообщаете Scala, что нужно, к примеру, чтобы тип `Queue[String]` рассматривался в качестве подтипа `Queue[AnyRef]`. Компилятор проверит факт определения `Queue` в соответствии со способом, предполагаемым подобным подтипированием.

Кроме префикса `+` существует префикс `-`, который показывает *контрвариантность* подтипирования. Если определение `Queue` имеет вид

```
trait Queue[-T] { … }
```

и, если тип `T` является подтипом типа `S`, это будет означать, что `Queue[S]` выступает подтипом `Queue[T]` (что в случае с очередями было бы довольно неожиданно!). Ковариантность, контрвариантность или невариантность параметра типа называются *вариантностью* параметров. Знаки `+` и `-`, которые могут размещаться рядом с параметрами типа, называются *аннотациями вариантности*.

В чисто функциональном мире многие типы по своей природе обладают ковариантностью (гибкостью). Но ситуация становится иной, как только появляются изменяемые данные. Чтобы выяснить, почему так происходит, рассмотрим показанный в листинге 19.5 простой тип одноэлементных ячеек, допускающих чтение и запись.

Листинг 19.5. Невариантный (жесткий) класс Cel

```
class Cell[T](init: T) {
  private[this] var current = init
  def get = current
```

```
  def set(x: T) = { current = x }
}
```

Тип `Cell`, показанный в листинге 19.5, объявлен невариантным (жестким). Ради аргументации представим на время, что `Cell` был объявлен ковариантным, то есть был объявлен класс `Cell[+T]` и этот код передан компилятору Scala. (Так делать не стоит, и скоро мы объясним, почему.) Затем можно сконструировать такую проблематичную последовательность инструкций:

```
val c1 = new Cell[String]("abc")
val c2: Cell[Any] = c1
c2.set(1)
val s: String = c1.get
```

Если рассмотреть строки по отдельности, то все они выглядят вполне нормально. В первой строке создается строковая ячейка, которая сохраняется в `val`-переменной по имени `c1`. Во второй строке определяется новая `val`-переменная `c2`, имеющая тип `Cell[Any]`, которая инициализируется значением переменной `c1`. Вроде все в порядке, поскольку экземпляры класса `Cell` считаются ковариантными. В третьей строке для `c2` устанавливается значение `1`. С этим тоже все в порядке, поскольку присваиваемое значение `1` является экземпляром, относящимся к объявленному для `c2` типу элемента `Any`. И наконец, в последней строке значение элемента `c1` присваивается строковой переменной. Здесь нет ничего странного, так как с обеих сторон выражения находятся значения одного и того же типа. Но если взять все это вместе, то четыре строки в конечном счете присваивают целочисленное значение `1` строковому значению `s`. Налицо явное нарушение целостности типа.

Какая из операций является причиной сбоя в ходе выполнения кода? Видимо, вторая, где используется ковариантное подтипирование. Все другие операции слишком просты и

основательны. Стало быть, ячейка `Cell`, хранящая значение типа `String`, не является также ячейкой `Cell`, хранящей значение типа `Any`, поскольку существуют вещи, которые можно делать с `Cell` из `Any`, но нельзя делать с `Cell` из `String`. К примеру, в отношении `Cell` из `String` нельзя использовать `set` с `Int`-аргументом.

Получается, если передать ковариантную версию `Cell` компилятору Scala, будет выдана ошибка компиляции:

```
Cell.scala:7: error: covariant type T occurs in
contravariant position in type T of value x

   def set(x: T) = current = x

               ^
```

Вариантность и массивы. Интересно сравнить это поведение с массивами в Java. В принципе, массивы подобны ячейкам, за исключением того, что у них может быть более одного элемента. При этом массивы в Java считаются ковариантными.

Пример, аналогичный работе с ячейкой, можно проверить в работе с массивами Java:

```
// Это код Java
String[] a1 = { "abc" };
Object[] a2 = a1;
a2[0] = new Integer(17);
String s = a1[0];
```

Если запустить в работу данный пример, то окажется, что он пройдет компиляцию. Но в ходе выполнения, когда элементу `a2[0]` будет присваиваться `Integer`-значение, программа выдаст исключение `ArrayStore`:

```
Exception in thread "main"
java.lang.ArrayStoreException:
```

```
java.lang.Integer
        at JavaArrays.main(JavaArrays.java:8)
```

Дело в том, что в Java тип элемента сохраняется в массиве в ходе выполнения программы. При каждом обновлении элемента массива новое значение элемента проверяется на принадлежность к сохраняемому в массиве типу. Если оно не является экземпляром этого типа, выдается исключение `ArrayStore`.

Может возникнуть вопрос: а почему в Java принята такая на вид небезопасная и затратная конструкция? Отвечая на него, Джеймс Гослинг (James Gosling), основной изобретатель языка Java, говорил, что создатели надеялись получить простые средства обобщенной обработки массивов. Например, им хотелось получить возможность написания метода сортировки всех элементов массива с использованием следующей сигнатуры, которая получает массив из `Object`-элементов:

```
void sort(Object[] a, Comparator cmp) { ... }
```

Ковариантность массивов нужна была для того, чтобы этому методу сортировки можно было передавать массивы произвольных ссылочных типов. Разумеется, после появления в Java обобщенных типов (дженериков) подобный метод сортировки уже мог быть написан с параметрами типа, поэтому необходимость в ковариантности массивов отпала. Сохранение ее до наших дней обусловлено соображениями совместимости.

Scala старается быть чище, чем Java, и не считает массивы ковариантными. Вот что получится, если перевести первые две строки кода примера с массивом на язык Scala:

```
scala> val a1 = Array("abc")
a1: Array[String] = Array(abc)

scala> val a2: Array[Any] = a1
<console>:8: error: type mismatch;
```

```
found : Array[String]
required: Array[Any]
       val a2: Array[Any] = a1
                            ^
```

В данном случае получилось, что Scala считает массивы невариантными (жесткими), следовательно, `Array[String]` не считается соответствующим `Array[Any]`. Но иногда необходимо организовать взаимодействие между имеющимися в Java устаревшими методами, использующими в качестве средства эмуляции обобщенного массива `Object`-массив. Например, может возникнуть потребность вызова метода сортировки вроде того, который был рассмотрен ранее в отношении `String`-массива, передаваемого в качестве аргумента. Чтобы допустить такую возможность, Scala позволяет выполнять приведение массива из `T`-элементов к массиву элементов любого из подтипов `T`:

```
scala> val a2: Array[Object] =
         a1.asInstanceOf[Array[Object]]
a2: Array[Object] = Array(abc)
```

Приведение типов в ходе компиляции не вызывает никаких возражений и всегда будет успешно работать в ходе выполнения программы, поскольку положенная в основу работы JVM модель времени выполнения рассматривает массивы в качестве ковариантных точно так же, как это делается в языке Java. Но впоследствии, как и на Java, может быть получено исключение `ArrayStore`.

19.4. Проверка аннотаций вариантности

После того как было рассмотрено несколько примеров ненадежности вариантности, можно задать вопрос: какие именно разновидности классов следует отвергать, а какие — принимать?

До сих пор во всех нарушениях целостности типов фигурировали поля или элементы массивов, которым переприсваивались значения. В то же время чисто функциональная реализация очередей представляется хорошим кандидатом на ковариантность. Но в следующем примере показано, что ситуацию ненадежности можно подстроить даже при отсутствии поля, которому переприсваивается значение.

Для выстраивания примера предположим, что очереди из листинга 19.4 определены как ковариантные. Затем создадим подкласс очередей с указанным типом `Int` и переопределим метод `enqueue`:

```
class StrangeIntQueue extends Queue[Int] {
  override def enqueue(x: Int) = {
    println(math.sqrt(x))
    super.enqueue(x)
  }
}
```

Перед выполнением добавления метод `enqueue` в подклассе `StrangeIntQueue` выводит на стандартное устройство квадратный корень из своего (целочисленного) аргумента.

Теперь можно создать контрпример из двух строк кода:

```
val x: Queue[Any] = new StrangeIntQueue
x.enqueue("abc")
```

Первая из этих двух строк вполне допустима, поскольку `StrangeIntQueue` является подклассом `Queue[Int]` и, если предполагается ковариантность очередей, `Queue[Int]` является подтипом `Queue[Any]`. Вторая строка также вполне допустима, поскольку `String`-значение можно добавлять к `Queue[Any]`. Но если взять их вместе, то у этих двух строк проявляется не имеющий никакого смысла эффект применения метода извлечения

квадратного корня к строке.

Это явно не те простые изменяемые поля, делающие ковариантные поля ненадежными. Проблема носит более общий характер. Получается, что как только обобщенный параметр типа появляется в качестве типа параметра метода, содержащие такой метод класс или трейт в данном параметре типа могут не быть ковариантными.

Для очередей метод `enqueue` нарушает это условие:

```
class Queue[+T] {
  def enqueue(x: T) =
    ...
}
```

При запуске модифицированного класса очередей, подобного показанному ранее, в компиляторе Scala он выдаст следующий результат:

```
Queues.scala:11: error: covariant type T occurs in
contravariant position in type T of value x
  def enqueue(x: T) =
              ^
```

Поля с переприсваиваемыми значениями являются частным случаем правила, не позволяющего параметрам типа, имеющим аннотацию +, использоваться в качестве типов параметра метода. Как упоминалось в разделе 18.2, поле с переприсваиваемым значением `var x: T` рассматривается в Scala как метод получения значения `def x: T` и как метод присваивания значения `def x_=(y: T)`. Как видите, метод присваивания значения имеет параметр с имеющимся у поля типом `T`. Следовательно, этот тип не может быть ковариантным.

Ускоренный режим чтения

Далее в этом разделе будет рассматриваться механизм, с помощью которого компилятор Scala проверяет аннотацию вариантности. Если эти подробности вас пока не интересуют, можете смело переходить к разделу 19.5. Следует усвоить главное: компилятор Scala будет проверять любую аннотацию вариантности, указанную вами в отношении параметров типа. Например, при попытке объявления ковариантного параметра типа (путем добавления знака +), способного вызвать потенциальные ошибки в ходе выполнения программы, программа откомпилирована не будет.

Для проверки правильности аннотаций вариантности компилятор Scala классифицирует все позиции в теле класса или трейта как положительные, отрицательные или нейтральные. Под позицией понимается любое место в теле класса или трейта (далее в тексте будет фигурировать только термин «класс»), где может использоваться параметр типа. Например, позицией является каждый параметр значения в методе, поскольку у параметра значения метода есть тип. Следовательно, в этой позиции может применяться параметр типа.

Компилятор проверяет использование всех имеющихся в классе параметров типа. Параметры типа, для аннотации которых используется знак +, могут применяться только в положительных позициях, а те, для аннотации которых используется знак -, — только в отрицательных позициях. Параметр типа, не имеющий аннотации вариантности, может использоваться в любой позиции, и, таким образом, он является единственной разновидностью параметров типа, которая может применяться в нейтральных позициях тела класса.

Для классифицирования позиций компилятор начинает работу с объявления параметра типа, а затем идет дальше через более глубокие уровни вложенности. Позиции на верхнем уровне

объявляемого класса определяются как положительные. По умолчанию позиции в более глубоких уровнях вложенности классифицируются таким же образом, как и охватывающие их уровни, но существует небольшое количество исключений, где классификация меняется. Позиции параметров значений метода классифицируются по *перевернутой* схеме относительно позиций за пределами метода, там положительная классификация становится отрицательной, отрицательная — положительной, а нейтральная позиция так и остается нейтральной.

Текущая классификация действует по перевернутой схеме в отношении не только позиций параметров значений методов, но и параметров типа методов. В зависимости от вариантности соответствующего параметра типа классификация иногда бывает перевернута в имеющейся в типе позиции аргумента типа, например это касается `Arg` в `C[Arg]`. Когда параметр типа у `C` аннотирован с применением знака `+`, классификация остается такой же. Когда параметр типа у `C` аннотирован с применением знака `-`, классификация определяется по перевернутой схеме. Если параметр типа у `C` не имеет аннотации вариантности, текущая классификация изменяется на нейтральную.

В качестве слегка надуманного примера рассмотрим следующее определение класса, где несколько позиций аннотированы согласно их классификации с помощью обозначений `^+` (для положительных) или `^-` (для отрицательных):

```
abstract class Cat[-T, +U] {
   def meow[W^-](volume: T^-, listener: Cat[U^+,
T^-]^-)
    : Cat[Cat[U^+, T^-]^-, U^+]^+
}
```

Позиции параметра типа `W` и двух параметров значений, `volume` и `listener`, помечены как отрицательные. Если посмотреть на тип результата метода `meow`, то позиция первого

аргумента, `Cat[U, T]`, помечена как отрицательная, поскольку первый параметр типа у `Cat`, `T`, аннотирован с использованием знака `-`. Тип `U` внутри этого аргумента опять имеет положительную позицию (после двух перевертываний), а тип `T` внутри этого аргумента остается в отрицательной позиции.

Из всего этого можно сделать вывод, что отследить позиции вариантностей очень трудно. Поэтому помощь компилятора Scala, проделывающего эту работу за вас, несомненно, приветствуется.

После вычисления классификаций компилятор проверяет, что каждый параметр типа используется только в позициях, имеющих соответствующую классификацию. В данном случае `T` используется только в отрицательных позициях, а `U` — только в положительных. Следовательно, класс `Cat` типизирован корректно.

19.5. Нижние ограничители

Вернемся к классу `Queue`. Вы видели, что прежнее определение `Queue[T]`, показанное в листинге 19.4, не может быть превращено в ковариантное в отношении `T`, поскольку `T` фигурирует в качестве типа параметра метода `enqueue` и находится в отрицательной позиции.

К счастью, существует способ открепиться: можно обобщить `enqueue`, превратив этот метод в полиморфный (то есть предоставить самому методу `enqueue` параметр типа), и воспользоваться *нижним ограничителем* его параметра типа. Новая формулировка `Queue` с реализацией этой идеи показана в листинге 19.6.

Листинг 19.6. Параметр типа с нижним ограничителем

```
class Queue[+T] (private val leading: List[T],
    private val trailing: List[T] ) {
```

```
  def enqueue[U >: T](x: U) =
    new Queue[U](leading, x :: trailing) // ...
}
```

В новом определении `enqueue` дается параметр типа `U`, и с помощью синтаксиса `U >: T` тип `T` определяется как нижний ограничитель для `U`. В результате от типа `U` требуется, чтобы он был родительским типом для `T`[112]. Теперь параметр для `enqueue` имеет тип `U`, а не тип `T`, а возвращаемое значение метода теперь не `Queue[T]`, а `Queue[U]`.

Предположим, к примеру, что существует класс `Fruit`, имеющий два подкласса, `Apple` и `Orange`. С новым определением класса `Queue` появилась возможность добавления `Orange` в `Queue[Apple]`. Результатом будет `Queue[Fruit]`.

В этом пересмотренном определении `enqueue` типы используются правильно. Интуитивно понятно, что, если `T` является более конкретным типом, чем ожидалось (например, `Apple` вместо `Fruit`), вызов `enqueue` все равно продолжит работать, поскольку `U` (`Fruit`) по-прежнему будет родительским типом для `T` (`Apple`)[113].

Возможно, новое определение `enqueue` лучше старого, поскольку оно имеет более обобщенный характер. В отличие от старой версии, новое определение позволяет добавлять к очереди с элементами типа `T` произвольный родительский тип `U`. Результат получается типа `Queue[U]`. Наряду с ковариантностью очереди это дает правильную разновидность гибкости для моделирования очередей из различных типов элементов вполне естественным образом. Тем самым показывается, что в совокупности аннотации вариантности и нижний ограничитель являются хорошо сыгранной командой. Они представляют отличный пример *ведения разработок на основе применяемых типов*, где типы интерфейса служат направляющими для детальной проработки конструкции и ее реализации. В случае с очередями высока вероятность того, что

вы не стали бы продумывать улучшенную реализацию enqueue с нижним ограничителем. Но у вас могло созреть решение сделать очереди ковариантными, в случае чего компилятор указал бы для enqueue на ошибку вариантности. Исправление ошибки вариантности путем добавления нижнего ограничителя придает enqueue большую обобщенность, а очереди в целом делает более полезными.

Это наблюдение является также главной причиной отданного в Scala предпочтения вариантности по месту объявления (declaration-site variance) над вариантностью по месту использования (use-site variance), встречающейся в Java в подстановочных символах или масках (wildcards). При использовании вариантности по месту использования вы разрабатываете класс самостоятельно. А вот клиентам этого класса придется вставлять подстановочные символы, и, если они сделают это неправильно, применение некоторых важных методов экземпляра станет невозможным. Вариантность — дело непростое, пользователи зачастую понимают ее неправильно и избегают ее, полагая, что маски и обобщения для них слишком сложны. При использовании вариантности по месту объявления ваши намерения выражаются для компилятора, который выполнит двойную проверку, чтобы убедиться, что метод, который вам нужно сделать доступным, был действительно доступен.

19.6. Контрвариантность

До сих пор во всех представленных в данной главе примерах встречалась либо ковариантность, либо невариантность. Но бывают такие случаи, при которых вполне естественно выглядит и контрвариантность. Рассмотрим, к примеру, трейт выходных каналов, показанный в листинге 19.7.

Листинг 19.7. Контрвариантный выходной канал

```
trait OutputChannel[-T] {
  def write(x: T)
}
```

Здесь трейт `OutputChannel` определен с контрвариантностью, указанной для `T`. Следовательно, получается, что выходной канал из `AnyRefs` является подтипом выходного канала из `Strings`. Хотя на интуитивном уровне это может показаться непонятным, в действительности в этом есть определенный смысл. Чтобы разобраться, почему так, посмотрим, что можно сделать с `OutputChannel[String]`. Единственной поддерживаемой операцией является запись в него значения типа `String`. Аналогичная операция может быть выполнена также в отношении `OutputChannel[AnyRef]`. Следовательно, вполне безопасно будет вместо `OutputChannel[String]` подставить `OutputChannel[AnyRef]`. В отличие от этого, подставить `OutputChannel[String]` туда, где требуется `OutputChannel[AnyRef]`, будет небезопасно. В конце концов, на `OutputChannel[AnyRef]` можно отправить любой объект, а `OutputChannel[String]` требует, чтобы все записываемые значения были строками.

Эти рассуждения указывают на общий принцип разработки систем типов: вполне безопасно предположить, что тип `T` является подтипом типа `U`, если значение типа `T` можно подставить там, где требуется значение типа `U`. Это называется принципом подстановки Лисков. Данный принцип соблюдается, если `T` поддерживает те же операции, что и `U`, и все принадлежащие `T` операции требуют меньшего, а предоставляют большее, чем соответствующие операции в `U`. В случае с выходными каналами `OutputChannel[AnyRef]` может быть подтипом `OutputChannel[String]`, поскольку в обоих типах поддерживается одна и та же операция `write` и эта операция требует меньшего в `OutputChannel[AnyRef]`, чем в

OutputChannel[String]. Меньшее означает, что от аргумента в первом случае требуется только, чтобы он был типа AnyRef, а вот во втором случае — чтобы он был типа String.

Иногда в одном и том же типе смешиваются ковариантность и контрвариантность. Известным примером являются функциональные трейты Scala. Например, при написании функционального типа A => B Scala раскрывает этот код, приводя его к виду Function1[A, B]. Определение Function1 в стандартной библиотеке использует как ковариантность, так и контрвариантность: как показано в листинге 19.8, трейт Function1 контрвариантен в аргументе функции типа S и ковариантен в типе результата T. Принцип подстановки Лисков здесь не нарушается, потому что аргументы — это то, что требуется, а вот результаты — то, что предоставляется.

Листинг 19.8. Ковариантность и контрвариантность Function1

```
trait Function1[-S, +T] {
  def apply(x: S): T }
```

Рассмотрим в качестве примера приложение, показанное в листинге 19.9. Здесь класс Publication содержит одно параметрическое поле title типа String. Класс Book расширяет Publication и пересылает свой строковый параметр title конструктору своего родительского класса. В синглтон-объекте Library определяются набор книг books и метод printBookList, получающий функцию info, имеющую тип Book => AnyRef. Иными словами, типом единственного параметра printBookList является функция, получающая один аргумент типа Book и возвращающая значение типа AnyRef. В приложении Customer определяется метод getTitle, получающий в качестве единственного своего параметра значение типа Publication и возвращающий значение типа String, содержащее название

переданной публикации Publication.

Листинг 19.9. Демонстрация вариантности параметра типа функции

```
class Publication(val title: String)
class Book(title: String) extends
Publication(title)
object Library {
  val books: Set[Book] =
    Set(
      new Book("Programming in Scala"),
      new Book("Walden")
    )
  def printBookList(info: Book => AnyRef) = {
    for (book <- books) println(info(book))
  }
}

object Customer extends App {
  def getTitle(p: Publication): String = p.title
  Library.printBookList(getTitle)
}
```

Теперь посмотрим на последнюю строку в объекте Customer. В этой строке вызывается принадлежащий Library метод printBookList, которому в инкапсулированном в значение функции виде передается getTitle:

```
Library.printBookList(getTitle)
```

Эта строка кода проходит проверку на соответствие типу даже при том, что String, тип результата выполнения функции, является подтипом AnyRef, типом результата параметра info

метода printBookList. Этот код проходит компилятор, потому что типы результатов функций объявлены ковариантными (+T в листинге 19.8). Если заглянуть в тело printBookList, можно получить представление о том, почему в этом есть определенный смысл.

Метод printBookList выполняет последовательный перебор элементов своего списка книг и вызывает переданную ему функцию в отношении каждой книги. Он пересылает AnyRef-результат, возвращенный info, методу println, который вызывает в отношении этого результата метод toString и выводит на стандартное устройство возвращенную им строку. Этот процесс будет работать со String-значениями, а также с любыми другими подклассами AnyRef, в чем, собственно, и заключается смысл ковариантности типов результатов функций.

Теперь рассмотрим параметр типа той функции, которая была передана методу printBookList. Хотя тип параметра, принадлежащего методу printBookList, объявлен как Book, функция getTitle при ее передаче в этот метод получает значение типа Publication, а этот тип является для Book *родительским типом*. Все это работает, потому что, хотя типом параметра метода printBookList является Book, телу метода printBookList разрешено лишь передать значение типа Book в функцию. А поскольку параметром типа функции getTitle является Publication, телу этой функции будет лишь разрешено обращаться к его параметру p, относящемуся к элементам, объявленным в классе Publication. Поскольку любой метод, объявленный в классе Publication, доступен также в его подклассе Book, все должно работать, в чем, собственно, и заключается смысл контрвариантности типов результатов функций. Графическое представление всего этого можно увидеть на рис. 19.1.

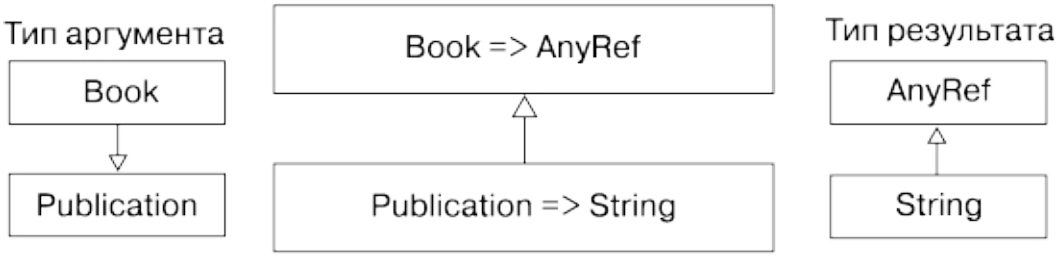


Рис. 19.1. Ковариантность и контрвариантность в параметрах типа функции

Код в листинге 19.9 проходит компиляцию, потому что `Publication => String` является подтипом `Book => AnyRef`, что и показано в центре схемы. Поскольку тип результата `Function1` определен в качестве ковариантного, представленное в правой части схемы взаимоотношение наследования двух типов результатов имеет то же самое направление, что и две функции, расположенные в центре. В отличие от этого, поскольку тип параметра функции `Function1` определен в качестве контрвариантного, взаимоотношение наследования двух типов параметров, показанное в левой части схемы, имеет направление, обратное направлению взаимоотношения наследования двух функций.

19.7. Закрытые данные объекта

У рассмотренного ранее класса `Queue` имеется проблема, заключающаяся в том, что операция `mirror` будет многократно копировать список `trailing` в список `leading`, если метод `head` вызывается несколько раз подряд в отношении списка, когда список `leading` пуст. Неэкономного копирования можно избежать, добавив некоторые рациональные побочные эффекты. В листинге 19.10 представлена новая реализация `Queue`, выполняющая не более одной коррекции списка `leading` за счет списка `trailing` для любой последовательности операций `head`.

Листинг 19.10. Оптимизированная функциональная очередь

```
class Queue[+T] private (
  private[this] var leading: List[T],
  private[this] var trailing: List[T]
) {

  private def mirror() =
    if (leading.isEmpty) {
      while (!trailing.isEmpty) {
        leading = trailing.head :: leading
        trailing = trailing.tail
      }
    }

  def head: T = {
    mirror()
    leading.head
  }
  def tail: Queue[T] = {
    mirror()
    new Queue(leading.tail, trailing)
  }

  def enqueue[U >: T](x: U) =
    new Queue[U](leading, x :: trailing)
}
```

В отличие от прежней версии, теперь `leading` и `trailing` являются переменными, которым можно переприсваивать значения, а `mirror` не возвращает новую очередь, а создает в отношении текущей в качестве побочного эффекта реверсную копию из `trailing` в `leading`. По отношению к реализации `Queue` этот побочный эффект носит чисто внутренний характер:

поскольку `leading` и `trailing` являются закрытыми переменными, клиентам `Queue` этот эффект не виден. Следовательно, согласно терминологии, установленной в главе 18, новая версия `Queue` просто определяет чисто функциональные объекты, несмотря на то что теперь в них содержатся поля с переприсваиваемыми значениями.

Прохождение этого кода через имеющийся в Scala механизм проверки типов может вызвать удивление. Все-таки у очередей теперь имеются два поля, которым можно переприсваивать значения ковариантного параметра типа `T`. Не нарушаются ли здесь правила вариантности? Может быть, нарушение и существовало бы, если бы не небольшая деталь: у `leading` и `trailing` имеется модификатор `private[this]`, стало быть, они объявлены закрытыми.

Как упоминалось в разделе 13.5, к закрытым элементам имеется доступ только из того объекта, в котором они определены. Получается, что обращение к переменным из того же объекта, в котором они определены, не вызывает проблем с вариантностью. Интуитивно понятное объяснение состоит в том, что для создания обстоятельств, при которых вариантность вызовет ошибки типа, нужно иметь ссылку на объект, содержащий статически более слабый тип, чем тот, с которым был определен объект. Что же касается обращений к закрытым значениям объекта, создать подобные обстоятельства невозможно.

Для определений закрытых элементов объекта в используемых в Scala правилах проверки вариантности есть исключение. Когда проверяется факт нахождения параметра типа с аннотацией либо `+`, либо `-` только в тех позициях, которые имеют такую же классификацию вариантности, такие определения пропускаются. Следовательно, код, показанный в листинге 19.10, компилируется без ошибок. В то же время, если в двух модификаторах `private` пропустить квалификатор `[this]`, будут показаны две ошибки типа:

```
Queues.scala:1: error: covariant type T occurs in
contravariant position in type List[T] of
parameter of
setter leading_=
class Queue[+T] private (private var leading:
List[T],
                                    ^

Queues.scala:1: error: covariant type T occurs in
contravariant position in type List[T] of
parameter of
setter trailing_=
                                private var trailing:
List[T]) {
                                    ^
```

19.8. Верхние ограничители

В листинге 16.1 была показана предназначенная для списков функция сортировки методом слияния, которая получала в качестве своего первого аргумента функцию сравнения, а в качестве второго, каррированного, аргумента — сортируемый список. Еще один способ, который может вам пригодиться для организации подобной функции сортировки, заключается в требовании того, чтобы тип списка подмешивался в трейт `Ordered`. Как упоминалось в разделе 12.4, подмешивание `Ordered` к классу и реализация в `Ordered` одного абстрактного метода, `compare`, позволят клиентам сравнивать экземпляры класса с помощью операторов <, >, <= и >=. В качестве примера в листинге 19.11 показан трейт `Ordered`, подмешанный к классу `Person`.

В результате двух людей можно сравнивать так:

```
scala> val robert = new Person("Robert", "Jones")
```

```
robert: Person = Robert Jones

scala> val sally = new Person("Sally", "Smith")
sally: Person = Sally Smith

scala> robert < sally
res0: Boolean = true
```

Листинг 19.11. Класс Person, к которому подмешан трейт Ordered

```
class Person(val firstName: String, val lastName:
String)
  extends Ordered[Person] {

  def compare(that: Person) = {
    val lastNameComparison =
      lastName.compareToIgnoreCase(that.lastName)
    if (lastNameComparison != 0)
      lastNameComparison
    else
      firstName.compareToIgnoreCase(that.firstName)
  }

   override def toString = firstName + " " +
lastName
}
```

Чтобы выставить требование о подмешивании типа списка, переданного вашей новой функции сортировки, в `Ordered`, следует воспользоваться верхним ограничителем. Верхний ограничитель указывается аналогично нижнему, за исключением того, что вместо обозначения `>:`, используемого для нижних ограничителей, применяется, как показано в листинге 19.12,

обозначение <:.

Листинг 19.12. Функция сравнения с верхним ограничителем

```
def    orderedMergeSort[T    <:    Ordered[T]](xs:
List[T]): List[T] = {
  def merge(xs: List[T], ys: List[T]): List[T] =
    (xs, ys) match {
      case (Nil, _) => ys
      case (_, Nil) => xs
      case (x :: xs1, y :: ys1) =>
        if (x < y) x :: merge(xs1, ys)
        else y :: merge(xs, ys1)
    }
  val n = xs.length / 2
  if (n == 0) xs
  else {
    val (ys, zs) = xs splitAt n
                        merge(orderedMergeSort(ys),
orderedMergeSort(zs))
  }
}
```

Используя синтаксис `T <: Ordered[T]`, вы показываете, что параметр типа `T` имеет верхний ограничитель `Ordered[T]`. Это означает, что тип элемента, переданного `orderedMergeSort`, должен быть подтипом `Ordered`. Следовательно, `List[Person]` можно передать `orderedMergeSort`, поскольку `Person` подмешивается в `Ordered`.

Рассмотрим, к примеру, следующий список:

```
scala> val people = List(
         new Person("Larry", "Wall"),
```

```
        new Person("Anders", "Hejlsberg"),
        new Person("Guido", "van Rossum"),
        new Person("Alan", "Kay"),
        new Person("Yukihiro", "Matsumoto")
      )
people: List[Person] = List(Larry Wall, Anders Hejlsberg,
  Guido van Rossum, Alan Kay, Yukihiro Matsumoto)
```

Поскольку тип элемента этого списка `Person` подмешивается к `Ordered[People]` (и поэтому является его подтипом), список можно передать методу `orderedMergeSort`:

```
scala> val sortedPeople = orderedMergeSort(people)
sortedPeople: List[Person] = List(Anders Hejlsberg, Alan Kay,
   Yukihiro Matsumoto, Guido van Rossum, Larry Wall)
```

А теперь следует заметить, что, хотя функция сортировки, показанная в листинге 19.12, служит неплохой иллюстрацией верхних ограничителей, в действительности это не самый универсальный способ в Scala для разработки функции сортировки, получающей преимущества от трейта `Ordered`.

Функцию `orderedMergeSort` нельзя, к примеру, использовать для сортировки списка целых чисел, поскольку класс `Int` не является подтипом `Ordered[Int]`:

```
scala> val wontCompile = orderedMergeSort(List(3, 2, 1))
<console>:5: error: inferred type arguments [Int] do
   not conform to method orderedMergeSort's type
      parameter bounds [T <: Ordered[T]]
```

```
        val wontCompile = orderedMergeSort(List(3,
2, 1))
                                  ^
```

Для получения более универсального решения в разделе 21.6 будет показан порядок использования подразумеваемых параметров и контекстных ограничителей.

Резюме

В этой главе был показан ряд технологий, применяемых для сокрытия информации: закрытые конструкторы, фабричные методы, абстракция типов и закрытые элементы объекта. Кроме этого, были рассмотрены способы указания вариантности типов данных и объяснено, что эта вариантность означает для реализации класса. И наконец, продемонстрированы две технологии, помогающие получить гибкие аннотации вариантности: нижние ограничители для параметров типов методов и аннотации `private[this]` для локальных полей и методов.

112 Взаимоотношения родительского и подчиненного типов носят возвратный характер. Это означает, что тип выступает для самого себя как родительским, так и подчиненным. Даже при том что T является для U нижней границей, T все же можно передавать методу enqueue.

113 С технической точки зрения произошедшее является переворотом для нижних границ. Параметр типа U находится в отрицательной позиции (один переворот), а нижняя граница (>: T) — в положительной позиции (два переворота).

20. Абстрактные элементы

Элемент класса или трейта называется *абстрактным*, если у него нет в классе полного определения. Реализовывать абстрактные элементы предполагается в подклассах того класса, в котором они объявлены. Воплощение этой идеи можно найти во многих объектно-ориентированных языках. Например, в Java можно объявить абстрактные методы. В Scala также можно объявить такие методы, что было показано в разделе 10.2. Но Scala этим не ограничивается, и в нем данная идея реализуется самым универсальным образом: в качестве элементов классов и трейтов можно объявлять не только методы, но и абстрактные поля и даже абстрактные типы.

В этой главе будут рассмотрены все четыре разновидности абстрактных элементов: `val`- и `var`-переменные, методы и типы. Попутно мы разберем предварительно инициализированные поля, «ленивые» `val`-переменные, типы, зависящие от пути и перечисления.

20.1. Краткий обзор абстрактных элементов

В следующем трейте объявляется по одному абстрактному элементу из каждой разновидности: абстрактный тип (`T`), метод (`transform`), `val`-переменная (`initial`) и `var`-переменная (`current`):

```
trait Abstract {
  type T
  def transform(x: T): T
  val initial: T
  var current: T
}
```

Конкретная реализация трейта `Abstract` нуждается в заполнении определений для каждого из его абстрактных элементов. Пример реализации, представляющий эти определения, выглядит следующим образом:

```
class Concrete extends Abstract {
  type T = String
  def transform(x: String) = x + x
  val initial = "hi"
  var current = initial
}
```

Реализация придает конкретное значение типу `T`, определяя его в качестве псевдонима типа `String`. Операция `transform` объединяет предоставленную ей строку с нею же самой, а для `initial`, как и для `current`, устанавливается значение `"hi"`.

Эти примеры дают вам первое приблизительное представление о том, какие разновидности абстрактных элементов существуют в Scala. Далее рассматриваются подробности, касающиеся этих элементов, и объясняется, для чего эти новые формы абстрактных элементов, а также элементы типа в целом могут пригодиться.

20.2. Элементы типа

В примере, приведенном в предыдущем разделе, было показано, что понятие *абстрактный тип* в Scala означает объявление типа (с ключевым словом `type`) в качестве элемента класса или трейта, без указания определения. Абстрактными могут быть и сами классы, а трейты по определению абстрактные, но ни один из них не является в Scala тем, что называют *абстрактным типом*. Абстрактный тип в Scala всегда является элементом какого-либо класса или трейта, как тип `T` в трейте `Abstract`.

Неабстрактный (или конкретный) элемент типа, такой как тип `T`

в классе Concrete, можно представить себе в качестве способа определения нового имени или *псевдонима* для типа. К примеру, в классе Concrete типу String дается псевдоним T. В результате везде, где в определении класса Concrete появляется T, подразумевается String. Сюда включаются преобразования типов параметров и результатов как исходных, так и текущих, в которых при их объявлении в родительском трейте Abstract упоминается T. Следовательно, когда в классе Concrete реализуются эти методы, такие обозначения T интерпретируются как String.

Одним из поводов использования элемента типа является определение краткого описательного псевдонима для типа, чье имя длиннее или значение менее понятно, чем у псевдонима. Такие элементы типа могут сделать понятнее код класса или трейта. Другим основным применением элементов типа является объявление абстрактного типа, который должен быть определен в подклассе. Подробнее этот вариант использования, продемонстрированный в предыдущем разделе, будет рассмотрен чуть позже.

20.3. Абстрактные val-переменные

Объявление абстрактной val-переменной выглядит следующим образом:

```
val initial: String
```

Val-переменной даются имя и тип, но не указывается значение. Это значение должно быть предоставлено конкретным определением val-переменной в подклассе. Например, в классе Concrete для реализации val-переменной используется следующий код:

```
val initial = "hi"
```

Объявление в классе абстрактной `val`-переменной применяется, когда в этом классе еще не известно нужное ей значение, но известно, что переменная в каждом экземпляре класса получит неизменяемое значение.

Объявление абстрактной `val`-переменной напоминает объявление абстрактного метода без параметров:

```
def initial: String
```

Клиентский код будет ссылаться как на `val`-переменную, так и на метод абсолютно одинаково (то есть `obj.initial`). Но если `initial` является абстрактной `val`-переменной, клиенту гарантируется, что `obj.initial` будет при каждом обращении выдавать одно и то же значение. Если идентификатор `initial` относится к абстрактному методу, эта гарантия соблюдаться не будет, поскольку в таком случае метод `initial` может быть реализован с помощью конкретного метода, возвращающего при каждом своем вызове разные значения.

Иными словами, `val`-переменная ограничивает свою допустимую реализацию: любая реализация должна быть определением `val`-переменной — она не может быть `var`- или `def`-определением. А вот объявления абстрактных методов могут быть реализованы как конкретными определениями методов, так и конкретными определениями `var`-переменных. Если взять абстрактный класс `Fruit`, показанный в листинге 20.1, то класс `Apple` будет допустимой реализацией подкласса, а класс `BadApple` — нет.

Листинг 20.1. Переопределение абстрактных val-переменных и методов без параметров

```
abstract class Fruit {
  val v: String // 'v' для значения
```

```
  def m: String // 'm' для метода
}

abstract class Apple extends Fruit {
  val v: String
    val m: String // нормально воспринимаемое
переопределение 'def' в 'val'
}

abstract class BadApple extends Fruit {
   def v: String // ОШИБКА: переопределять 'val' в
'def' нельзя
  def m: String
}
```

20.4. Абстрактные var-переменные

Как и для абстрактной `val`-переменной, для абстрактной `var`-переменной объявляются только имя и тип, а начальное значение не указывается. Например, в листинге 20.2 показан трейт `AbstractTime`, в котором объявляются две абстрактные переменные с именами `hour` и `minute`.

Что означают такие абстрактные `var`-переменные, как `hour` и `minute`? В разделе 18.2 было показано, что `var`-переменные, объявленные в качестве элементов класса, оснащаются методами присваивания и получения значения. Это справедливо и для абстрактных `var`-переменных. Если, к примеру, объявляется абстрактная `var`-переменная по имени `hour`, то подразумевается, что для нее объявляется абстрактный метод получения значения `hour` и абстрактный метод присваивания значения `hour_=`. Тем самым не определяется никакого поля с возможностью переприсваивания значения, а конкретная реализация абстрактной `var`-переменной будет выполнена в подклассах. Например,

определение AbstractTime, показанное в листинге 20.2, абсолютно эквивалентно определению, показанному в листинге 20.3.

Листинг 20.2. Объявление абстрактных var-переменных

```
trait AbstractTime {
  var hour: Int
  var minute: Int
}
```

Листинг 20.3. Расширение абстрактных var-переменных в методы получения и присваивания значения

```
trait AbstractTime {
  def hour: Int          // get-метод для 'hour'
  def hour_=(x: Int)     // set-метод для 'hour'
  def minute: Int        // get-метод для 'minute'
  def minute_=(x: Int)   // set-метод для 'minute'
}
```

20.5. Инициализация абстрактных val-переменных

Иногда абстрактные val-переменные играют роль, аналогичную роли параметров родительского класса: они позволяют предоставить в подклассе подробности, не указанные в родительском классе. На практике это обстоятельство играет важную роль для трейтов, поскольку у них нет конструкторов, которым можно передавать параметры. Таким образом, обычный принцип параметризации трейта работает через абстрактные val-переменные, реализованные в подклассах.

Рассмотрим в качестве примера переделку класса Rational из

главы 6, который был показан в листинге 6.5, в трейт:

```
trait RationalTrait {
  val numerArg: Int
  val denomArg: Int
}
```

У класса `Rational` из главы 6 было два параметра: `n` для числителя рационального числа и `d` для его знаменателя. Представленный здесь трейт `RationalTrait` определяет вместо них две абстрактные `val`-переменные: `numerArg` и `denomArg`. Чтобы создать конкретный экземпляр этого трейта, нужно реализовать определения абстрактных `val`-переменных, например:

```
new RationalTrait {
  val numerArg = 1
  val denomArg = 2
}
```

Здесь появляется ключевое слово `new`, после которого в фигурных скобках стоит тело трейта `RationalTrait`. Это выражение выдает экземпляр *безымянного класса*, подмешивающего трейт и определяемого с помощью тела. Создание экземпляра данного безымянного класса дает эффект, аналогичный созданию экземпляра с помощью кода `new Rational(1, 2)`.

Но аналогия здесь неполная. Существует небольшое различие, касающееся порядка, в котором инициализируются выражения. При записи следующего кода:

```
new Rational(expr1, expr2)
```

два выражения, `expr1` и `expr2`, вычисляются перед инициализацией класса `Rational`, следовательно, значения `expr1`

и expr2 доступны для инициализации класса Rational.

С трейтами складывается обратная ситуация. При записи кода: new RationalTrait {
  val numerArg = expr1
  val denomArg = expr2
}

выражения expr1 и expr2 вычисляются как часть инициализации безымянного класса, но безымянный класс инициализируется после RationalTrait. Следовательно, значения numerArg и denomArg в ходе инициализации RationalTrait недоступны (точнее говоря, выбор любого значения даст значение по умолчанию для типа Int, то есть нуль). Для представленного ранее определения RationalTrait это не проблема, поскольку при инициализации трейта значения numerArg или denomArg не используются. Но проблема возникает в варианте RationalTrait, показанном в листинге 20.4, где определяются обычные числители и знаменатели.

Листинг 20.4. Трейт, использующий абстрактные val-переменные

```
trait RationalTrait {
  val numerArg: Int
  val denomArg: Int
  require(denomArg != 0)
  private val g = gcd(numerArg, denomArg)
  val numer = numerArg / g
  val denom = denomArg / g
  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
  override def toString = numer + "/" + denom
}
```

При попытке создания экземпляра трейта с какими-либо выражениями для числителя и знаменателя, не являющимися простыми литералами, выдается исключение:

```
scala> val x = 2
x: Int = 2
scala> new RationalTrait {
         val numerArg = 1 * x
         val denomArg = 2 * x
       }
java.lang.IllegalArgumentException: requirement failed
  at scala.Predef$.require(Predef.scala:207)
  at RationalTrait$class.$init$(<console>:10)
  ... 28 elided
```

Исключение в этом примере было выдано потому, что при инициализации класса `RationalTrait` у `denomArg` сохранилось исходное нулевое значение, из-за чего вызов `require` завершился сбоем.

В этом примере продемонстрировано, что порядок инициализации для параметров класса и абстрактных полей разный. Аргумент параметра класса вычисляется до его передачи конструктору класса (если только это не параметр до востребования). В отличие от этого, реализация определения `val`-переменной, которая находится в подклассе, вычисляется только после инициализации родительского класса.

После выяснения причин, по которым поведение абстрактных `val`-переменных отличается от поведения параметров, было бы неплохо узнать, как с этим справиться. Можно ли определить `RationalTrait`, который может быть инициализирован действенным образом без опасения, что возникнут ошибки из-за неинициализированных полей? В действительности в Scala предлагаются два альтернативных решения этой проблемы:

предварительно проинициализированные поля и *ленивые val-переменные*. Эти решения рассматриваются в остальной части раздела.

Предварительно проинициализированные поля

Первое решение — предварительно проинициализированные поля — позволяет инициализировать поле подкласса до вызова родительского класса. Для этого нужно просто поместить определение поля в фигурные скобки перед вызовом конструктора класса. В качестве примера в листинге 20.5 показана еще одна попытка создания экземпляра `RationalTrait`. Из этого примера видно, что инициализирующая часть стоит перед упоминанием родительского трейта `RationalTrait`. Они разделены ключевым словом `with`.

Листинг 20.5. Предварительно проинициализированные поля в выражении безымянного класса

```
scala> new {
         val numerArg = 1 * x
         val denomArg = 2 * x
       } with RationalTrait
res1: RationalTrait = 1/2
```

Сфера применения предварительно проинициализированных полей не ограничивается безымянными классами, они могут использоваться также в объектах или именованных подклассах. Соответствующие примеры показаны в листингах 20.6 и 20.7. Из них видно, что в каждом случае раздел предварительной инициализации ставится после ключевого слова `extends`, относящегося к определяемому объекту или классу. Класс `RationalClass`, показанный в листинге 20.7, иллюстрирует

общую схему доступности параметров класса для инициализации родительского трейта.

Листинг 20.6. Предварительно проинициализированные поля в определении объекта

```
object twoThirds extends {
  val numerArg = 2
  val denomArg = 3
} with RationalTrait
```

Листинг 20.7. Предварительно проинициализированные поля в определении класса

```
class RationalClass(n: Int, d: Int) extends {
  val numerArg = n
  val denomArg = d } with RationalTrait {
  def + (that: RationalClass) = new RationalClass(
    numer * that.denom + that.numer * denom,
    denom * that.denom
  )
}
```

Поскольку инициализация рассматриваемых полей происходит до того, как вызывается конструктор родительского класса, их инициализаторы не могут ссылаться на создаваемый объект. Следовательно, если такой инициализатор ссылается на `this`, то происходит обращение к объекту, содержащему класс, или к объекту, который уже был создан, а не к самому создаваемому объекту.

Рассмотрим пример:

```
scala> new {
```

```
        val numerArg = 1
        val denomArg = this.numerArg * 2
      } with RationalTrait
<console>:11: error: value numerArg is not a
member of object
$iw
                  val denomArg = this.numerArg * 2
                                         ^
```

Пример не проходит компиляцию, поскольку ссылка вида `this.numerArg` нацеливалась на поле `numerArg` в объекте, содержащем `new` (в качестве которого в данном случае выступал синтезированный объект по имени `$iw`, в который интерпретатор помещает строки пользовательского ввода). Итак, еще раз: предварительно проинициализированные поля ведут себя в этом смысле подобно аргументам конструктора класса.

Ленивые val-переменные

Предварительно проинициализированные поля могут использоваться для точной имитации поведения инициализации аргументов конструктора класса. Но иногда лучше позволить самой системе разобраться, как и что должно быть проинициализировано. Добиться этого можно путем определения ленивой `val`-переменной. Если перед определением `val`-переменной поставить модификатор `lazy`, выражение инициализации в правой стороне будет вычисляться только при первом использовании `val`-переменной.

Определим, к примеру, объект `Demo` с `val`-переменной:

```
scala> object Demo {
             val x = { println("initializing x");
"done" }
       }
```

```
defined object Demo
```

Теперь сначала сошлемся на Demo, а затем на Demo.x:

```
scala> Demo
initializing x
res3: Demo.type = Demo$@2129a843

scala> Demo.x
res4: String = done
```

Как видите, на момент использования объекта Demo его поле x становится проинициализированным. Инициализация x составляет часть инициализации Demo. Но ситуация изменится, если определить поле x в качестве ленивого:

```
scala> object Demo {
         lazy val x = { println("initializing x");
"done" }
       }
defined object Demo

scala> Demo
res5: Demo.type = Demo$@5b1769c

scala> Demo.x
initializing x
res6: String = done
```

Теперь инициализация Demo не включает инициализацию x. Такая инициализация будет отложена до первого использования x. Это похоже на ситуацию определения x в качестве метода без параметров с использованием ключевого слова def. Но, в отличие от def, ленивая val-переменная никогда не вычисляется более

одного раза. Фактически после первого вычисления ленивой val-переменной результат вычисления сохраняется, чтобы его можно было применить повторно при последующем использовании той же самой val-переменной.

При изучении данного примера создается впечатление, что объекты, подобные Demo, сами ведут себя как ленивые val-переменные, поскольку они инициализируются по необходимости при их первом использовании. Так оно и есть. Действительно, определение объекта может рассматриваться как сокращенная запись для задания ленивой val-переменной с безымянным классом, в котором дается описание содержимого объекта.

Используя ленивые val-переменные, можно переделать RationalTrait, как показано в листинге 20.8. В новом определении трейта все конкретные поля заданы как ленивые. Еще одно изменение, касающееся предыдущего определения RationalTrait, показанного в листинге 20.4, заключается в том, что условие require было перемещено из тела трейта в инициализатор закрытого поля g, вычисляющий наибольший общий делитель для numerArg и denomArg. После внесения этих изменений при инициализации LazyRationalTrait делать больше ничего не нужно, поскольку весь код инициализации теперь является правосторонней частью ленивой val-переменной. Таким образом, вполне безопасно инициализировать абстрактные поля LazyRationalTrait после того, как класс уже определен.

Листинг 20.8. Инициализация трейта с ленивыми val-переменными

```
trait LazyRationalTrait {
  val numerArg: Int
  val denomArg: Int
  lazy val numer = numerArg / g
  lazy val denom = denomArg / g
  override def toString = numer + "/" + denom
```

```
  private lazy val g = {
    require(denomArg != 0)
    gcd(numerArg, denomArg)
  }
  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
}
```

Рассмотрим пример:

```
scala> val x = 2
x: Int = 2

scala> new LazyRationalTrait {
         val numerArg = 1 * x
         val denomArg = 2 * x
       }
res7: LazyRationalTrait = 1/2
```

Какой-либо предварительной инициализации здесь не требуется. Поучительно будет проследить последовательность инициализации, приводящей к тому, что в показанном ранее коде на стандартное устройство будет выведена строка `1/2`:

Создается новый экземпляр `LazyRationalTrait`, и запускается код инициализации `LazyRationalTrait`. Этот код пуст — ни одно из полей `LazyRationalTrait` еще не проинициализировано.

10. Вычислением выражения `new` определяется первичный конструктор безымянного подкласса. Эта процедура включает в себя инициализацию `numerArg` значением `2` и инициализацию `denomArg` значением `4`.

11. Далее интерпретатор в отношении создаваемого объекта вызывает метод `toString`, чтобы получившееся значение можно было бы вывести на стандартное устройство.

12. Метод `toString`, определенный в трейте `LazyRationalTrait`, выполняет первое обращение к полю `numer`, что вызывает вычисление инициализатора.

13. Инициализатор поля `numer` обращается к закрытому полю `g`, следовательно, следующим вычисляется `g`. При этом вычислении происходит обращение к `numerArg` и `denomArg`, которые были определены в шаге 2.

14. Метод `toString` обращается к значению `denom`, что вызывает вычисление `denom`. При этом происходит обращение к значениям `denomArg` и `g`. Инициализатор поля `g` заново уже не вычисляется, поскольку он был вычислен в шаге 5.

15. Создается и выводится строка результата `1/2`.

Заметьте, что в классе `LazyRationalTrait` определение `g` появляется в тексте кода после определений в нем `numer` и `denom`. Несмотря на это, поскольку все три значения ленивые, `g` проходит инициализацию до завершения инициализации `numer` и `denom`.

Тем самым демонстрируется важное свойство ленивых `val`-переменных: порядок следования их определений в тексте кода не играет никакой роли, поскольку инициализация значений выполняется по требованию. Стало быть, ленивые `val`-переменные могут освободить вас как программиста от необходимости обдумывания порядка расстановки определений `val`-переменных, чтобы гарантировать, что к моменту своей востребованности все будет определено.

Но это преимущество сохраняется до тех пор, пока

инициализация ленивых `val`-переменных не производит никаких побочных эффектов, а также не зависит от них. Если есть побочные эффекты, то порядок инициализации становится значимым. И тогда могут возникнуть серьезные трудности в отслеживании порядка запуска инициализационного кода, как было показано в предыдущем примере. Следовательно, ленивые `val`-переменные являются идеальным компонентом функциональных объектов, где порядок инициализации не имеет значения до тех пор, пока все в конечном счете не будет проинициализировано. А вот для преимущественно императивного кода эти переменные подходят меньше.

> **Ленивые функциональные языки**
>
> Scala — далеко не первый язык, использующий идеальную пару ленивых определений и функционального кода. Существует целая категория ленивых языков функционального программирования, в которых каждое значение и каждый параметр проходят ленивую инициализацию. Ярким представителем этого класса языков является Haskell1.

20.6. Абстрактные типы[114]

В начале этой главы в качестве объявления абстрактного типа был показан код `type T`. Далее мы рассмотрим, что означает такое объявление абстрактного типа и для чего оно может пригодиться. Как и все остальные объявления абстракций, объявление абстрактного типа является заместителем для чего-либо, что будет конкретно определено в подклассах. В данном случае это тип, который мы определим ниже по иерархии классов. Следовательно, обозначение `T` ссылается на тип, который на момент его

объявления еще неизвестен. Разные подклассы могут обеспечивать различные реализации T.

Рассмотрим широко известный пример, в который абстрактные типы вписываются вполне естественно. Предположим, что получена задача моделирования привычного рациона животных. Приступить к ее решению можно с определения класса питания Food и класса животных Animal с методом питания eat:

```
class Food
abstract class Animal {
  def eat(food: Food)
}
```

Затем можно попробовать создать специализацию этих двух классов в виде класса коров Cow, питающихся травой Grass:

```
class Grass extends Food
class Cow extends Animal {
   override def eat(food: Grass) = {} // Этот код
не пройдет компиляцию
}
```

Но при попытке компилирования этих новых классов будут получены следующие ошибки компиляции:

```
BuggyAnimals.scala:7: error: class Cow needs to be
abstract, since method eat in class Animal of type
    (Food)Unit is not defined
class Cow extends Animal {
      ^
BuggyAnimals.scala:8: error: method eat overrides
nothing
  override def eat(food: Grass) = {}
               ^
```

Дело в том, что метод `eat` в классе `Cow` не переопределяет метод `eat` в классе `Animal`, потому что типы их параметров различаются: в классе `Cow` это `Grass`, а в классе `Animal` это `Food`.

Встречаются утверждения, что в отклонении этих двух классов виновата слишком строгая система типов. При этом говорится, что было бы неплохо уточнить параметр методов в подклассе. Но если бы классы были позволены в том виде, в котором они написаны, вы быстро оказались бы в весьма небезопасной для себя ситуации.

К примеру, механизму проверки типов будет передан следующий сценарий:

```
class Food
abstract class Animal {
  def eat(food: Food)
}
class Grass extends Food
class Cow extends Animal {
   override def eat(food: Grass) = {} // Этот код
не пройдет компиляцию,
}                                     // но, если
бы это случилось...
class Fish extends Food
val bessy: Animal = new Cow
bessy eat (new Fish)     // ...коров можно было бы
накормить рыбой.
```

Если снять ограничения, программа пройдет компиляцию, поскольку коровы из класса `Cow` являются животными из класса `Animal`, а у класса `Animal` имеется метод кормления `eat`, который принимает любую разновидность питания `Food`, включая рыбу, то есть `Fish`. Но коровы не едят рыбу!

Вместо этого вам нужно применить более точное моделирование. Животные из класса `Animal` потребляют (`eat`)

питание Food, но какое именно питание потребляет каждое животное, зависит от самого животного. Это довольно четко можно выразить с помощью абстрактного типа, что и показано в листинге 20.9.

Листинг 20.9. Моделирование подходящего питания с помощью абстрактных типов

```
class Food
abstract class Animal {
  type SuitableFood <: Food
  def eat(food: SuitableFood)
}
```

С новым определением класса животное Animal может потреблять только то питание, которое ему подходит. Какое именно питание будет подходящим, не может быть определено на уровне класса Animal. Поэтому подходящее питание SuitableFood моделируется в виде абстрактного типа. У типа есть верхний ограничитель Food, что выражено условием <: Food. Это означает, что любая конкретная реализация SuitableFood (в подклассе класса Animal) должна быть подклассом Food. К примеру, реализовать SuitableFood с классом IOException не получится.

После определения Animal можно, как показано в листинге 20.10, перейти к коровам. Класс Cow устанавливает в качестве подходящего для коров питания SuitableFood траву Grass, а также определяет конкретный метод eat для данной разновидности питания.

Листинг 20.10. Реализация абстрактного типа в подклассе

```
class Grass extends Food
class Cow extends Animal {
  type SuitableFood = Grass
  override def eat(food: Grass) = {}
}
```

Эти новые определения класса компилируются без ошибок. При попытке запуска с новыми определениями класса контрпримера про коров, которые едят рыбу (cows-that-eat-fish), будут получены следующие ошибки компиляции:

```
scala> class Fish extends Food
defined class Fish
scala> val bessy: Animal = new Cow
bessy: Animal = Cow@1515d8a6

scala> bessy eat (new Fish)
<console>:14: error: type mismatch;
found : Fish
required: bessy.SuitableFood
              bessy eat (new Fish)
                        ^
```

20.7. Типы, зависящие от пути

Давайте еще раз посмотрим на последнее сообщение об ошибке. Нас интересует тип, требующийся для метода `eat`: `bessy.SuitableFood`. Указание типа состоит из ссылки на объект, `bessy`, за которой следует поле типа объекта, `SuitableFood`. Тем самым показывается, что объекты в Scala в качестве элементов могут иметь типы. Смысл `bessy.SuitableFood` раскрывается так: тип `SuitableFood`, являющийся элементом объекта по ссылке `bessy`, или, иначе, тип питания, подходящего для `bessy`.

Тип вида `bessy.SuitableFood` называется *типом, зависящим от пути* (path-dependent type). Слово «путь» здесь означает ссылку на объект. Он может быть в виде единственного имени, такого как `bessy`, или более длинного пути доступа, такого как `farm.barn.bessy`, где все составляющие, `farm`, `barn` и `bessy`, являются переменными (или именами синглтон-объектов), ссылающимися на объекты.

В соответствии со значением понятия «тип, зависящий от пути» этот тип зависит от пути, различные пути дают начало разным типам. Например, предположим, что для определения классов собачьего питания `DogFood` и собак `Dog` используется следующий код:

```
class DogFood extends Food
class Dog extends Animal {
  type SuitableFood = DogFood
  override def eat(food: DogFood) = {}
}
```

При попытке накормить собаку едой для коров ваш код не пройдет компиляцию:

```
scala> val bessy = new Cow
bessy: Cow = Cow@713e7e09

scala> val lassie = new Dog
lassie: Dog = Dog@6eaf2c57

scala> lassie eat (new bessy.SuitableFood)
<console>:16: error: type mismatch;
found : Grass
required: DogFood
              lassie eat (new bessy.SuitableFood)
```

```
                                                    ^
```

Проблема заключается в том, что типом объекта `SuitableFood`, переданного методу `eat`, является `bessy.SuitableFood`, а он несовместим с параметром типа `eat`, которым является `lassie.SuitableFood`.

Обстоятельства для двух собак из класса `Dog` будут разными. Поскольку в классе `Dog` тип `SuitableFood` определен в качестве псевдонима для класса `DogFood`, типы `SuitableFood` двух представителей класса `Dogs` по факту одинаковы. В результате тот экземпляр класса `Dog`, который называется `lassie`, фактически может питаться тем, что подходит другому экземпляру класса `Dog`, который мы назовем `bootsie`:

```
scala> val bootsie = new Dog
bootsie: Dog = Dog@13a7c48c

scala> lassie eat (new bootsie.SuitableFood)
```

Тип, зависящий от пути, напоминает синтаксис для типа внутреннего класса в Java, но есть существенное различие: в типе, зависящем от пути, называется внешний объект, а в типе внутреннего класса — внешний класс. Типы внутренних классов в стиле Java могут быть выражены и в Scala, но записываются они по-другому. Рассмотрим два класса, наружный `Outer` и внутренний `Inner`:

```
class Outer {
  class Inner
}
```

В Scala вместо используемого в Java выражения `Outer.Inner` к внутреннему классу обращаются с помощью выражения `Outer#Inner`. Синтаксис с использованием точки «.»

зарезервирован для объектов. Представим себе, к примеру, что создаются экземпляры двух объектов типа `Outer`:

```
val o1 = new Outer
val o2 = new Outer
```

Здесь `o1.Inner` и `o2.Inner` являются двумя типами, зависящими от пути, и это разные типы. Оба этих типа соответствуют более общему типу `Outer#Inner` (являются его подтипами), представляющему класс `Inner` с произвольным внешним объектом типа `Outer`. В отличие от этого, тип `o1.Inner` ссылается на класс `Inner` с определенным внешним объектом, на который ссылается `o1`. Точно так же тип `o2.Inner` ссылается на класс `Inner` с другим определенным внешним объектом, на который ссылается `o2`.

В Scala, как и в Java, экземпляры внутреннего класса содержат ссылку на экземпляр охватывающего их внешнего класса. Это, к примеру, позволяет внутреннему классу обращаться к элементам его внешнего класса. Таким образом, невозможно добавить экземпляр внутреннего класса без какого-либо способа указания экземпляра внешнего класса. Один из способов заключается в создании экземпляра внутреннего класса внутри тела внешнего класса. В этом случае будет использован текущий экземпляр внешнего класса (в ссылке на который можно задействовать `this`).

Еще один способ заключается в использовании типа, зависящего от пути. Например, поскольку в типе `o1.Inner` присутствует название определенного внешнего объекта, вы можете создать его экземпляр:

```
scala> new o1.Inner
res11: o1.Inner = Outer$Inner@1ae1e03f
```

Получившийся внутренний объект будет содержать ссылку на свой внешний объект, то есть на объект, на который ссылается `o1`.

В отличие от этого, поскольку тип `Outer#Inner` не содержит названия какого-либо определенного экземпляра класса `Outer`, создать экземпляр этого класса невозможно:

```
scala> new Outer#Inner
<console>:9: error: Outer is not a legal prefix for a constructor
              new Outer#Inner
                          ^
```

20.8. Уточняющие типы

Когда класс является наследником другого класса, первый класс называют *номинальным* подтипом другого класса. Этот подтип номинальный, поскольку у каждого типа есть имя и имена явным образом объявлены имеющими взаимоотношения подтипирования. В Scala также дополнительно поддерживается структурное подтипирование, где взаимоотношения подтипирования возникают просто потому, что у двух типов есть совместимые элементы. Для получения в Scala структурного подтипирования нужно воспользоваться имеющимися в этом языке *уточняющими типами*.

Обычно удобнее применять номинальное типирование, поэтому в любой новой конструкции нужно сначала попробовать воспользоваться именно этим типированием. Имя является одним кратким идентификатором, следовательно, оно короче явного перечисления типов элементов. Кроме того, структурное подтипирование зачастую проявляет более гибкий характер, чем вам нужно. Тем не менее оно имеет свои преимущества. Одно из них заключается в том, что иногда действительно не нужно ничего определять в виде типов, кроме элементов самого класса. Предположим, к примеру, что нужно определить класс пастбища `Pasture`, в котором могут быть животные, поедающие траву. Одним из вариантов может быть определение трейта для

животных, питающихся травой, — AnimalThatEatsGrass, и его подмешивание в каждый класс, где он применяется. Но это будет слишком многословным решением. В классе Cow уже имеется объявление, что это животное и оно ест траву, а теперь придется объявлять, что это еще и животное, поедающее траву.

Вместо определения AnimalThatEatsGrass можно воспользоваться уточняющим типом. Просто запишите основной тип, Animal, а за ним последовательность элементов, перечисленную в фигурных скобках. Элементы в фигурных скобках представляют дальнейшие указания, или, если хотите, уточнения типов элементов из основного класса.

Вот как записывается тип «животное, поедающее траву»:

```
Animal { type SuitableFood = Grass }
```

Теперь, имея в своем распоряжении этот тип, класс пастбища можно записать следующим образом:

```
class Pasture {
   var animals: List[Animal { type SuitableFood =
Grass }] = Nil
  // ...
}
```

20.9. Перечисления

Интересный вариант применения типов, зависящих от пути, можно найти в имеющейся в Scala поддержке перечислений. В некоторых других языках, включая Java и C#, перечисления фигурируют в качестве встроенных в язык конструкций для определения новых типов. Scala не нуждается в специальном синтаксисе для перечислений. Вместо этого в стандартной библиотеке языка имеется класс scala.Enumeration.

Для создания нового перечисления задается объект,

расширяющий этот класс. В следующем примере показано определение нового перечисления под названием Colors:

```
object Color extends Enumeration {
  val Red = Value
  val Green = Value
  val Blue = Value
}
```

Scala позволяет также сократить ряд последовательных определений val- или var-переменных, у которых правая сторона выражений одинаковая. В качестве эквивалента показанного ранее кода можно сделать следующую запись:

```
object Color extends Enumeration {
  val Red, Green, Blue = Value
}
```

В этом определении объекта предоставляются три значения: Color.Red, Color.Green и Color.Blue. Также все, что имеется в Color, можно импортировать с помощью следующего кода:

```
import Color._
```

а затем просто использовать имена Red, Green и Blue. Но каким будет тип этих значений?

Enumeration определяет внутренний класс по имени Value, а метод без параметров с таким же именем Value возвращает новый экземпляр этого класса. Иными словами, типом такого значения, как Color.Red, является Color.Value, который является типом всех перечисляемых значений, определенных в объекте Color. Это тип, зависящий от пути, где Color является путем, а Value — зависящим от него типом. Важно, что это совершенно новый тип, отличающийся от других типов.

В частности, если определить еще одно перечисление:

```
object Direction extends Enumeration {
  val North, East, South, West = Value
}
```

то `Direction.Value` будет отличаться от `Color.Value`, поскольку та часть, которая относится к пути, у этих двух типов разная.

Имеющийся в Scala класс `Enumeration` предлагает также многие другие свойства, которые можно найти в конструкциях перечислений иных языков. Со значениями перечислений можно связать имена, воспользовавшись для этого другим перегружаемым вариантом метода `Value`:

```
object Direction extends Enumeration {
  val North = Value("North")
  val East = Value("East")
  val South = Value("South")
  val West = Value("West")
}
```

Значения в перечислении можно перебрать через набор, возвращенный имеющимся в перечислении методом `values`:

```
scala> for (d <- Direction.values) print(d + " ")
North East South West
```

Значения в перечислении нумеруются с нуля, а номер значения в перечислении можно определить с помощью имеющегося в нем метода `id`:

```
scala> Direction.East.id
res14: Int = 1
```

Также можно пойти другим путем — от неотрицательного целого числа, которое служит этому номеру идентификатором в

перечислении:

```
scala> Direction(1)
res15: Direction.Value = East
```

Этого должно быть вполне достаточно, чтобы приступить к работе с перечислениями. Дополнительные сведения можно получить в комментариях Scaladoc для класса `scala.Enumeration`.

20.10. Практический пример: работа с валютой

Далее в главе рассмотрен практический пример, объясняющий порядок использования в Scala абстрактных типов. При этом будет поставлена задача разработки класса `Currency`. Обычный экземпляр `Currency` будет представлять денежную сумму в долларах, евро, йенах и некоторых других валютах. Он позволит совершать с валютой ряд арифметических операций. Например, можно будет сложить две суммы в одной и той же валюте. Или умножить текущую сумму на коэффициент процентной ставки.

Эти замыслы приводят к следующей первой конструкции класса валют:

```
// Первая (нерабочая) конструкция класса Currency
abstract class Currency {
  val amount: Long
  def designation: String
    override def toString = amount + " " +
designation
  def + (that: Currency): Currency = ...
  def * (x: Double): Currency = ...
}
```

Поле `amount` (сумма) в классе валют является количеством

представляемых ею валютных единиц. Это поле имеет тип Long, то есть представляемая сумма денежных средств может быть очень крупной, сравнимой с рыночной капитализацией Google или Apple. Здесь оно оставлено абстрактным в ожидании своего определения, когда в подклассе зайдет речь о конкретной сумме. Наименование валюты designation является строкой, которая идентифицирует эту валюту. Метод toString класса Currency показывает сумму и наименование валюты. Он будет выдавать результат следующего вида:

```
79 USD
11000 Yen
99 Euro
```

И наконец, имеются метод + для сложения сумм в валюте и метод * для умножения суммы в валюте на число с плавающей точкой. Конкретное значение в валюте можно создать, предоставив конкретные значения суммы и наименования валюты:

```
new Currency {
  val amount = 79L
  def designation = "USD"
}
```

Эта конструкция не вызовет нареканий, если задумано моделирование с использованием только одной валюты, например только долларов или только евро. Но она не будет работать, если понадобится иметь дело сразу с несколькими валютами. Предположим, выполняется моделирование долларов и евро в качестве двух подклассов класса валюты:

```
abstract class Dollar extends Currency {
  def designation = "USD"
}
abstract class Euro extends Currency {
```

```
  def designation = "Euro"
}
```

На первый взгляд все выглядит вполне разумно. Но данный код позволит складывать доллары с евро. Результатом такого сложения окажется тип `Currency`. Но это будет весьма забавная валюта — смесь евро и долларов. Вместо этого нужно получить более специализированную версию метода `+`. При его реализации в классе `Dollar` он должен получать аргументы типа `Dollar` и выдавать результат типа `Dollar`; при реализации в классе `Euro` — получать аргументы типа `Euro` и выдавать результат типа `Euro`. Следовательно, тип метода сложения будет изменяться в зависимости от того, в каком классе он находится. И все же хотелось бы создать метод сложения единожды, а не делать это при каждом новом определении валюты.

Чтобы справиться с подобными ситуациями, в Scala имеется весьма простая технология. Если к моменту определения класса что-то еще неизвестно, сделайте это что-то абстрактным. Такое действие применимо как к значениям, так и к типам. Для валют точные типы аргументов и результата метода сложения неизвестны, следовательно, они являются подходящими кандидатами для того, чтобы стать абстрактными.

Это привело бы к появлению следующей предварительной версии кода класса `AbstractCurrency`:

```
// Вторая (все еще несовершенная) конструкция
класса Currency
abstract class AbstractCurrency {
  type Currency <: AbstractCurrency
  val amount: Long
  def designation: String
    override def toString = amount + " " +
designation
  def + (that: Currency): Currency = ...
```

```
  def * (x: Double): Currency = ...
}
```

Единственное отличие от прежней ситуации заключается в том, что класс теперь называется `AbstractCurrency` и содержит абстрактный тип `Currency`, представляющий стоящую под вопросом реальную валюту. Каждому конкретному подклассу `AbstractCurrency` придется фиксировать тип `Currency` для обозначения конкретного подкласса как такового, тем самым затягивая узел.

Вот как, к примеру, выглядит новая версия класса `Dollar`, которая теперь расширяет класс `AbstractCurrency`:

```
abstract class Dollar extends AbstractCurrency {
  type Currency = Dollar
  def designation = "USD"
}
```

Эта конструкция вполне работоспособна, но по-прежнему далека от совершенства. Одна проблема скрывается за многоточиями, которые показывают в классе `AbstractCurrency` пропущенные определения методов `+` и `*`. В частности, как в этом классе должен быть реализован метод сложения? Нетрудно вычислить правильную сумму в новой валюте как `this.amount + that.amount`. Но как преобразовать сумму в валюту нужного типа?

Можно попробовать применить следующий код:

```
def + (that: Currency): Currency = new Currency {
  val amount = this.amount + that.amount
}
```

Но он не пройдет компиляцию:

```
error: class type required
  def + (that: Currency): Currency = new Currency
```

```
{
                                                    ^
```

Одно из ограничений в трактовке в языке Scala абстрактных типов заключается в невозможности создания экземпляра абстрактного типа, а также в невозможности абстрактного типа играть роль родительского типа для другого класса[115]. Следовательно, компилятор будет отвергать код показанного здесь примера, в котором предпринимается попытка создания экземпляра `Currency`.

Но эти ограничения можно обойти, используя фабричный метод. Вместо создания экземпляра абстрактного класса напрямую объявите абстрактный метод, занимающийся выполнением этой задачи. Затем там, где абстрактный тип устанавливается в какой-либо конкретный тип, вам также нужно предоставить конкретную реализацию фабричного метода. Для класса `AbstractCurrency` это будет выглядеть следующим образом:

```
abstract class AbstractCurrency {
  type Currency <: AbstractCurrency // абстрактный
тип
   def make(amount: Long): Currency  // фабричный
метод
   ...                                          // вся
остальная часть определения класса
}
```

Подобную конструкцию, конечно, можно заставить работать, но выглядит она как-то подозрительно. Зачем помещать фабричный метод внутрь класса `AbstractCurrency`? Это выглядит довольно сомнительно как минимум по двум причинам. Во-первых, если имеется некоторая сумма в валюте (скажем, один доллар), то у вас также есть возможность нарастить сумму в этой же валюте, используя следующий код:

```
myDollar.make(100) // здесь еще сто!
```

В эпоху цветных ксероксов этот сценарий может стать заманчивым, но следует надеяться, что никто не сможет проделывать это слишком долго, не будучи пойманным за руку. Во-вторых, этот код, если у вас уже есть ссылка на объект Currency, позволяет создавать дополнительные объекты Currency. Но как получить первый объект данной валюты Currency? Вам понадобится другой метод создания, выполняющий практически ту же работу, что и make. То есть вы столкнулись со случаем дублирования кода, являющимся верным признаком кода с душком.

Решение, конечно же, будет заключаться в перемещении абстрактного типа и фабричного класса за пределы класса AbstractCurrency. Нужно создать другой класс, содержащий класс AbstractCurrency, тип Currency и фабричный метод make.

Назовем этот класс CurrencyZone:

```
abstract class CurrencyZone {
  type Currency <: AbstractCurrency
  def make(x: Long): Currency
  abstract class AbstractCurrency {
    val amount: Long
    def designation: String
       override def toString = amount + " " +
designation
    def + (that: Currency): Currency =
      make(this.amount + that.amount)
    def * (x: Double): Currency =
      make((this.amount * x).toLong)
  }
}
```

Примером конкретизации CurrencyZone является объект US, который может быть определен следующим образом:

```
object US extends CurrencyZone {
  abstract class Dollar extends AbstractCurrency {
    def designation = "USD"
  }
  type Currency = Dollar
  def make(x: Long) = new Dollar { val amount = x
}
}
```

Здесь US является объектом, расширяющим CurrencyZone. В нем определяется класс Dollar, являющийся подклассом AbstractCurrency. Следовательно, типом денежных единиц в этой зоне является доллар США, US.Dollar. Объект US также устанавливает, что тип Currency будет псевдонимом для Dollar, и предоставляет реализацию фабричного метода make для возвращения суммы в долларах.

Конструкция вполне работоспособна. Можно лишь добавить несколько уточнений. Первое из них касается разменных монет. До сих пор каждая валюта измерялась в целых единицах: долларах, евро или йенах. Но у большинства валют имеются разменные монеты, например в США есть доллары и центы. Наиболее простым способом моделирования центов является использование поля amount в US.Currency, представленного в центах, а не в долларах. Чтобы вернуться к доллару, будет полезно ввести в класс CurrencyZone поле CurrencyUnit, содержащее одну стандартную единицу в данной валюте:

```
class CurrencyZone {
  ...
  val CurrencyUnit: Currency
}
```

Как показано в листинге 20.11, в объекте US могут быть заданы величины Cent, Dollar и CurrencyUnit.

Листинг 20.11. Зона валюты США

```
object US extends CurrencyZone {
  abstract class Dollar extends AbstractCurrency {
    def designation = "USD"
  }
  type Currency = Dollar
  def make(cents: Long) = new Dollar {
    val amount = cents
  }
  val Cent = make(1)
  val Dollar = make(100)
  val CurrencyUnit = Dollar
}
```

Это определение похоже на предыдущее определение объекта US, за исключением того, что в него добавлены три новых поля. Поле Cent представляет сумму в 1 US.Currency. Это объект, аналогичный одноцентовой монете. Поле Dollar представляет сумму в 100 US.Currency. Следовательно, объект US теперь определяет имя Dollar двумя способами. Тип Dollar, определенный абстрактным внутренним классом по имени Dollar, представляет общее название валюты Currency, действительное в валютной зоне US. В отличие от этого, значение Dollar, на которое ссылается val-поле по имени Dollar, представляет 1 доллар США, аналогичный однодолларовой купюре. Третье определение поля CurrencyUnit указывает на то, что стандартной денежной единицей в зоне США является доллар, Dollar, то есть значение Dollar, на которое ссылается поле, не является типом Dollar.

Метод toString в классе Currency также нуждается в адаптации для восприятия разменных монет на счету. Например, сумма 10 долларов и 23 цента должна выводиться как десятичное число: 10.23 USD. Чтобы добиться этого результата, принадлежащий Currency метод toString можно реализовать следующим образом:

```
override def toString =
                        ((amount.toDouble          /
CurrencyUnit.amount.toDouble)
   formatted ("%." + decimals(CurrencyUnit.amount)
+ "f")
   + " " + designation)
```

Здесь formatted является методом, доступным в Scala в нескольких классах, включая Double[116]. Метод formatted возвращает строку, полученную в результате форматирования исходной строки, в отношении которой был вызван метод formatted, в соответствии со строкой форматирования, переданной методу formatted в виде его правого операнда. Синтаксис строк форматирования, передаваемых методу formatted, аналогичен синтаксису, используемому для Java-метода String.format.

Например, строка форматирования %.2f приводит к выдаче числа с двумя знаками после точки. Строка форматирования, примененная в показанном ранее методе toString, собирается путем вызова метода decimals в отношении CurrencyUnit.amount. Этот метод возвращает число десятичных знаков десятичной степени за вычетом единицы. Например, decimals(10) — это 1, decimals(100) — это 2 и т. д. Метод decimals реализован в виде простой рекурсии:

```
private def decimals(n: Long): Int =
  if (n == 1) 0 else 1 + decimals(n / 10)
```

В листинге 20.12 показаны некоторые другие валютные зоны. В качестве еще одного уточнения к модели можно добавить свойство обмена валют. Сначала, как показано в листинге 20.13, можно создать объект Converter, содержащий применяемые обменные курсы валют. Затем к классу Currency можно добавить метод обмена from, который выполняет конвертацию из заданной исходной валюты в текущий объект Currency:

```
def from(other: CurrencyZone#AbstractCurrency):
Currency =
  make(math.round(
    other.amount.toDouble * Converter.exchangeRate
      (other.designation)(this.designation)))
```

Листинг 20.12. Валютные зоны для Европы и Японии

```
object Europe extends CurrencyZone {
  abstract class Euro extends AbstractCurrency {
    def designation = "EUR"
  }
  type Currency = Euro
  def make(cents: Long) = new Euro {
    val amount = cents
  }
  val Cent = make(1)
  val Euro = make(100)
  val CurrencyUnit = Euro
}

object Japan extends CurrencyZone {
  abstract class Yen extends AbstractCurrency {
    def designation = "JPY"
```

```
  }
  type Currency = Yen
  def make(yen: Long) = new Yen {
    val amount = yen
  }
  val Yen = make(1)
  val CurrencyUnit = Yen
}
```

Листинг 20.13. Объект converter с отображением курсов обмена

```
object Converter {
  var exchangeRate = Map(
    "USD" -> Map("USD" -> 1.0   , "EUR" -> 0.7596,
                 "JPY" -> 1.211 , "CHF" -> 1.223),
    "EUR" -> Map("USD" -> 1.316 , "EUR" -> 1.0   ,
                 "JPY" -> 1.594 , "CHF" -> 1.623),
    "JPY" -> Map("USD" -> 0.8257, "EUR" -> 0.6272,
                 "JPY" -> 1.0   , "CHF" -> 1.018),
    "CHF" -> Map("USD" -> 0.8108, "EUR" -> 0.6160,
                 "JPY" -> 0.982 , "CHF" -> 1.0 )
  )
}
```

Метод `from` получает в качестве аргумента произвольную валюту. Это выражено его формальным типом параметра `CurrencyZone#AbstractCurrency`, который показывает, что переданный как `other` аргумент должен быть типа `AbstractCurrency` в некоторой произвольной и неизвестной валютной зоне `CurrencyZone`. Результатом метода является произведение суммы в другой валюте и курса обмена между другой и текущей валютами. (Кстати, если вы полагаете, что сделка по

японской йене будет неудачной, курсы обмена валют основаны на числовых показателях в их CurrencyZone. Таким образом, 1,211 является курсом обмена центов США на японскую йену.)

Финальная версия класса `CurrencyZone` показана в листинге 20.14. Класс можно протестировать в командной оболочке Scala.

Предполагается, что класс `CurrencyZone` и все конкретные объекты `CurrencyZone` определены в пакете `org.stairwaybook.currencies`. Сперва нужно импортировать в командную оболочку `org.stairwaybook.currencies._`. Затем можно будет выполнить ряд обменных операций:

```
scala> Japan.Yen from US.Dollar * 100
res16: Japan.Currency = 12110 JPY

scala> Europe.Euro from res16
res17: Europe.Currency = 75.95 EUR

scala> US.Dollar from res17
res18: US.Currency = 99.95 USD
```

Листинг 20.14. Полный код класса CurrencyZone

```
abstract class CurrencyZone {

  type Currency <: AbstractCurrency
  def make(x: Long): Currency

  abstract class AbstractCurrency {

    val amount: Long
    def designation: String
```

```
    def + (that: Currency): Currency =
      make(this.amount + that.amount)
    def * (x: Double): Currency =
      make((this.amount * x).toLong)
    def - (that: Currency): Currency =
      make(this.amount - that.amount)
    def / (that: Double) =
      make((this.amount / that).toLong)
    def / (that: Currency) =
      this.amount.toDouble / that.amount

                                def       from(other:
CurrencyZone#AbstractCurrency): Currency =
      make(math.round(
                             other.amount.toDouble   *
Converter.exchangeRate
          (other.designation)(this.designation)))

    private def decimals(n: Long): Int =
      if (n == 1) 0 else 1 + decimals(n / 10)

    override def toString =
                               ((amount.toDouble     /
CurrencyUnit.amount.toDouble)
                              formatted    ("%."    +
decimals(CurrencyUnit.amount) + "f")
       + " " + designation)
  }

  val CurrencyUnit: Currency
}
```

Из факта получения почти такого же значения после трех конвертаций следует, что у нас весьма выгодные курсы обмена! Можно также нарастить значение в некоторой валюте:

```
scala> US.Dollar * 100 + res18
res19: US.Currency = 199.95 USD
```

В то же время складывать суммы разных валют нельзя:

```
scala> US.Dollar + Europe.Euro
<console>:12: error: type mismatch;
found : Europe.Euro
required: US.Currency
    (which expands to) US.Dollar
              US.Dollar + Europe.Euro
                                  ^
```

Абстракция типа выполняет свою работу, не позволяя складывать два значения в разных единицах измерений (в данном случае валютах). Она мешает нам выполнять необоснованные вычисления. Неверные преобразования между различными единицами могут показаться тривиальными недочетами, но они способны привести к весьма серьезным системным сбоям. Например, к аварии спутника Mars Climate Orbiter 23 сентября 1999 года, вызванной тем, что одна команда инженеров использовала метрическую систему мер, а другая — систему мер, принятую в Великобритании. Если бы единицы измерений были запрограммированы так же, как сделано с валютой в этой главе, эта ошибка была бы выявлена простым запуском кода на компиляцию. Но вместо этого она стала причиной аварии космического аппарата после почти десятимесячного полета.

Резюме

В Scala предлагается рационально структурированная и самая общая поддержка объектно-ориентированной абстракции. При этом допускается применение не только абстрактных методов, но и значений, переменных и типов. В этой главе были показаны способы извлечения преимуществ из использования абстрактных элементов класса. С их помощью реализуется простой, но весьма эффективный принцип структурирования систем: все неизвестное при разработке класса нужно превращать в абстрактные элементы. Тогда система типов задаст направление. И неважно, что именно будет неизвестно — тип, метод, переменная или значение. Все это в Scala может быть объявлено абстрактным.

114 Simon Peyton Jones, et. al. Haskell 98 Language and Libraries, Revised Report. Technical report // http://www.haskell.org/onlinereport, 2002.

115 В последнее время появились многообещающие исследования виртуальных классов, которые позволили бы сделать это, но пока виртуальные классы в Scala не поддерживаются.

116 Чтобы обеспечить доступность метода formatted, в Scala используются обогащающие оболочки, рассмотренные в разделе 5.10.

21. Подразумеваемые преобразования и параметры

Между вашим собственным кодом и библиотеками других разработчиков существует принципиальная разница: свой код при желании можно изменить или расширить, но, если нужно воспользоваться чьими-либо библиотеками, зачастую приходится воспринимать их такими, какие они есть. Чтобы смягчить эту проблему, в языках программирования появился ряд конструкций. В Ruby есть модули, а Smalltalk дает возможность пакетам добавляться к классам друг друга. Это, конечно, очень эффективно, но и довольно опасно, поскольку позволяет изменять поведение класса для всего приложения, о некоторых частях которого вам может быть ничего не известно. В C# 3.0 имеются статически расширяемые методы, имеющие более локальную, но и более ограниченную природу, позволяющую добавить к классу только методы, но не поля, и не позволяющую реализовывать в классе новые интерфейсы.

В Scala в ответ на это были введены скрытые преобразования и параметры. Они позволяют сделать существующие библиотеки гораздо приятнее для работы, давая возможность не указывать скучные, очевидные подробности, затмевающие интересные части вашего кода. При разумном использовании этого свойства получается код, сфокусированный на оригинальных, нетривиальных частях вашей программы. В этой главе показано, как работают подразумеваемые преобразования и параметры, и рассмотрены некоторые наиболее распространенные способы их применения.

21.1. Подразумеваемые преобразования

Перед тем как подробнее изучать подразумеваемые (или неявные)

преобразования, рассмотрим типичный пример их использования. Подразумеваемые преобразования зачастую удобны при работе с двумя основными частями программного средства, которые разрабатывались без учета друг друга. В каждой библиотеке имеются собственные способы решений одной и той же задачи, являющиеся, по сути, одним и тем же. Подразумеваемые преобразования помогают сократить количество необходимых явных преобразований из одного типа в другой.

Для реализации кроссплатформенных пользовательских интерфейсов в Java включена библиотека по имени Swing. Одна из выполняемых ею задач заключается в обработке событий от операционной системы, преобразовании их в независимые от используемой платформы объекты событий и передаче этих событий тем частям приложения, которые называются *отслеживателями событий* (event listeners).

Если бы Swing создавалась с прицелом на Scala, отслеживатели событий, наверное, были бы представлены типом функции. Тогда вызывающие объекты могли бы использовать синтаксис литерала функции в качестве облегченного способа указания того, что должно произойти для определенного класса событий. Поскольку в Java отсутствуют литералы функций, в Swing используется следующая наилучшая альтернатива — внутренний класс, реализующий интерфейс, состоящий из одного метода. В случае с отслеживателями действий (action listeners) интерфейс называется `ActionListener`.

Без использования подразумеваемого преобразования программа на Scala, которая применяет Swing, должна подобно программе на Java воспользоваться внутренними классами. Рассмотрим пример, создающий кнопку и подвешивающий на нее отслеживатель действий. Этот отслеживатель вызывается при щелчке на кнопке, после чего выводит строку `"pressed!"`:

```
val button = new JButton
button.addActionListener(
```

```
    new ActionListener {
      def actionPerformed(event: ActionEvent) = {
        println("pressed!")
      }
    }
  )
```

В этом коде много невыразительного, неинформативного содержимого. То, что это отслеживатель `ActionListener`, что метод обратного вызова называется `actionPerformed` и аргумент — `ActionEvent`, подразумевается для любого аргумента `addActionListener`. Единственной новой информацией здесь является выполняемый код, а именно вызов функции `println`. Эта новая информация заглушается шаблоном. Читатели кода должны обладать пронзительным орлиным взглядом, чтобы пробиться сквозь шум и обнаружить информативную часть.

Более подходящая для Scala версия получит в качестве аргумента функцию, существенно сокращая объем шаблонного кода:

```
button.addActionListener( // Несоответствие типов!
  (_: ActionEvent) => println("pressed!")
)
```

В данном виде этот код работать не будет[117]. Метод `addActionListener` требует отслеживателя действий, а получает функцию. Но, используя подразумеваемое преобразование, этот код можно привести в рабочее состояние.

Сначала нужно создать подразумеваемое преобразование между двумя типами. Это преобразование из функций в отслеживатели действий выглядит так:

```
implicit def function2ActionListener(f:
ActionEvent => Unit) =
```

```
  new ActionListener {
       def actionPerformed(event: ActionEvent) =
f(event)
  }
```

Это метод с одним аргументом, получающий функцию и возвращающий отслеживатель действий. Как и любой другой метод с одним аргументом, он может быть вызван напрямую, а его результат — передан другому выражению:

```
button.addActionListener(
  function2ActionListener(
    (_: ActionEvent) => println("pressed!")
  )
)
```

Это уже лучше версии с внутренним классом. Заметьте, как необоснованно большой объем шаблонного кода был сведен к замене литералом функции и вызову метода. Улучшить код удалось благодаря использованию подразумеваемого преобразования. Поскольку метод `function2ActionListener` помечен ключевым словом `implicit`, он может быть опущен и компилятор вставит его автоматически. В результате получится следующий код:

```
// Теперь все работает
button.addActionListener(
  (_: ActionEvent) => println("pressed!")
)
```

Этот код работает благодаря тому, что компилятор сначала пробует откомпилировать все как есть, но видит ошибку типа. Прежде чем окончательно сдаться, он ищет подразумеваемое преобразование, которое смогло бы исправить ситуацию. В данном случае он находит `function2ActionListener`, пробует

применить этот метод преобразования, видит, что он работает, и движется дальше. Здесь компилятор старается сделать так, чтобы разработчик мог проигнорировать еще одну волокитную деталь. Что попробовать, отслеживатель действий или функцию события действия? Либо одно, либо другое сработает, и он использует то, что больше подходит.

В данном разделе были продемонстрированы небольшая эффективность подразумеваемых преобразований и то, как они позволяют вам воспользоваться существующими библиотеками. В следующих разделах мы рассмотрим правила, определяющие, когда выполняются подразумеваемые преобразования и как они будут найдены.

21.2. Правила для подразумеваемых преобразований

К подразумеваемым определениям относятся те, которые компилятору разрешено вставлять в программу для устранения любой из ее ошибок типов. Например, если `x + y` не проходит проверку типов, компилятор может изменить этот код, чтобы получилось `convert(x) + y`, где под `convert` понимается некое доступное подразумеваемое преобразование. Если `convert` переделывает `x` во что-то, имеющее метод `+`, то эта переделка может исправить программу, чтобы она прошла проверку типов и выполнялась корректно. Если `convert` — всего лишь простая функция преобразования, то избавление от нее исходного кода может сделать его понятнее.

Подразумеваемые преобразования регулируются следующими общими правилами.

- **Правило маркировки:** доступны только определения с пометкой `implicit`. Ключевое слово `implicit` применяется для маркировки того объявления, которое компилятор может использовать в качестве подразумеваемого. Им можно помечать любое объявление переменной, функции или объекта. Пример

подразумеваемого объявления функции выглядит следующим образом[118]:

```
implicit def intToString(x: Int) = x.toString
```

Если `convert` имеет пометку `implicit`, компилятор всего лишь превратит `x + y` в `convert(x) + y`. Тем самым устраняется путаница, которая может возникнуть при выборе компилятором случайных функций, оказавшихся в области видимости, и вставке их в качестве преобразований. Свой выбор компилятор будет делать исходя только лишь из тех определений, у которых имеется явная пометка `implicit`.

- **Правило области видимости:** вставляемое подразумеваемое преобразование должно находиться в области видимости в качестве однокомпонентного идентификатора или быть связанным с исходным или целевым типом преобразования. Компилятор Scala будет рассматривать только те подразумеваемые преобразования, которые находятся в области видимости. Поэтому, чтобы подразумеваемое преобразование было доступно, нужно каким-то образом поместить его в область видимости. Более того, за единственным исключением, подразумеваемое преобразование должно находиться в области видимости в качестве *однокомпонентного идентификатора*. Компилятор не будет вставлять преобразование в форме `someVariable.convert`. Например, он не станет расширять `x + y` в `someVariable.convert(x) + y`. Если нужно сделать идентификатор `someVariable.convert` доступным в качестве подразумеваемого преобразования, его нужно будет импортировать, что сделает его доступным в качестве однокомпонентного идентификатора. После импортирования компилятор сможет свободно применить его как `convert(x) + y`. Фактически в библиотеки зачастую включают объект

Preamble, содержащий ряд полезных подразумеваемых преобразований. После этого в коде, использующем библиотеку, можно будет для обращения к имеющимся в библиотеке подразумеваемым преобразованиям однократно воспользоваться выражением import Preamble._.

В правиле однокомпонентного идентификатора есть одно исключение. Компилятор будет искать подразумеваемые определения или ожидаемые целевые типы преобразования также в объекте-спутнике исходного кода. Например, при попытке передачи объекта типа Dollar методу, получающему Euro, типом источника является Dollar, а целевым типом — Euro. Поэтому можно запаковать подразумеваемое преобразование из Dollar в Euro в объект-спутник любого класса, Dollar или Euro.

Пример, в котором подразумеваемое преобразование помещено в объект-спутник класса Dollar, выглядит следующим образом:

```
object Dollar {
  implicit def dollarToEuro(x: Dollar): Euro = ...
}
class Dollar { ... }
```

В данном случае преобразование dollarToEuro считается связанным с типом Dollar. Компилятор найдет такое связанное преобразование при каждой необходимости выполнения преобразования из экземпляра типа Dollar. Отдельно импортировать преобразование в вашу программу нет никакой необходимости.

Правило области видимости помогает обосновать применение модулей. Чтобы при чтении из файла последний отличался от других файлов, нужно учесть, что он либо должен быть импортирован, либо на него должна иметься явная ссылка с

полным указанием имени. Для подразумеваемых преобразований польза от этого не менее важна, чем для явно написанного кода. Если подразумеваемые преобразования распространяются на всю систему, то, чтобы понять, какой файл нужен, следует знать о каждом подразумеваемом преобразовании, используемом в программе!

- **Правило применения в конкретный момент времени только одной возможности:** может быть вставлено только одно подразумеваемое преобразование. Компилятор не станет переопределять `x + y` в `convert1(convert2(x)) + y`. Подобные действия приведут к резкому увеличению времени компиляции ошибочного кода и увеличат разницу между тем, что пишет программист, и тем, что фактически делает программа. Чтобы сохранить здравый смысл, компилятор не вставляет дополнительные подразумеваемые преобразования, когда он уже на полпути к применению другого подразумеваемого преобразования. Но это ограничение можно обойти за счет наличия у подразумеваемого преобразования подразумеваемых параметров, речь о которых пойдет чуть позже.

- **Правило приоритетности применения явно обозначенного кода:** когда код в том виде, в котором он записан, проходит проверку типов, не делаются попытки применения подразумеваемых преобразований. Компилятор не станет изменять уже работающий код. Следствием из этого правила является возможность замены подразумеваемого преобразования явно указанным. Это удлиняет код, но устраняет его очевидную двусмысленность. В каждом отдельно взятом случае можно менять один вариант на другой. Если код становится повторяющимся и многословным, то от однообразия поможет избавиться подразумеваемое преобразование. Если же краткость кода делает его непонятным, то преобразования можно указать явно.

Названия подразумеваемых преобразований

У подразумеваемых преобразований могут быть произвольные имена. Имя подразумеваемого преобразования играет определенную роль только в двух ситуациях: если его нужно записать явным образом в способе применения и если нужно определить, какие подразумеваемые преобразования доступны в том или ином месте программы. Чтобы проиллюстрировать вторую ситуацию, предположим, что имеется объект с двумя подразумеваемыми преобразованиями:

```
object MyConversions {
  implicit def stringWrapper(s: String):
    IndexedSeq[Char] = ...
  implicit def intToString(x: Int): String = ...
}
```

Вам нужно в создаваемом приложении воспользоваться преобразованием `stringWrapper` и совсем не нужно, чтобы целочисленные значения автоматически конвертировались в строки посредством преобразования `intToString`. Этого можно добиться импортированием только одного преобразования из двух:

```
import MyConversions.stringWrapper
... // код воспользуется stringWrapper
```

В этом примере важно, чтобы у подразумеваемых преобразований были имена, поскольку это единственный способ выполнить импортирование выборочно, применив только одно из двух преобразований.

Где применяются подразумеваемые элементы

Подразумеваемые элементы применяются в языке в трех местах: в преобразованиях в ожидаемый тип, в преобразованиях получателя

выбора и в подразумеваемых параметрах. Подразумеваемые преобразования в ожидаемый тип позволяют использовать один тип в том контексте, где ожидается другой тип. Например, можно располагать значением типа `String` и иметь желание передать его методу, требующему значения типа `IndexedSeq[Char]`. Преобразования получателя позволяют адаптировать получатель вызова метода (то есть объект, в отношении которого метод был вызван), если метод непригоден для исходного типа. Примером может послужить выражение `"abc".exists`, которое в результате преобразования приобретает вид `stringWrapper("abc").exists`, поскольку метод `exists` недоступен в классе `Strings`, но доступен в классе `IndexedSeqs`. Что же касается подразумеваемых параметров, то они обычно используются для предоставления вызываемой функции более подробной информации о том, что именно нужно вызывающему объекту. Подразумеваемые параметры особенно хорошо подходят для использования с обобщенными функциями, где вызываемая функция порой вообще ничего не знает о типе одного или нескольких аргументов. Все эти разновидности подразумеваемых элементов будут рассмотрены в следующих разделах.

21.3. Подразумеваемое преобразование в ожидаемый тип

Подразумеваемое преобразование в ожидаемый тип является первым местом, в котором компилятор будет использовать подразумеваемые элементы. Правило простое. Там, где компилятор видит *X*, а нужен *Y*, он станет искать подразумеваемую функцию, выполняющую преобразование *X* в *Y*. Например, обычно значение типа `Double` не может использоваться в качестве целочисленного значения, поскольку в нем будет потеряна точность:

```
scala> val i: Int = 3.5
<console>:7: error: type mismatch;
```

```
found : Double(3.5)
required: Int
       val i: Int = 3.5
                    ^
```

Но чтобы сгладить ситуацию, можно определить подразумеваемое преобразование:

```
scala> implicit def doubleToInt(x: Double) = x.toInt
doubleToInt: (x: Double)Int

scala> val i: Int = 3.5
i: Int = 3
```

Компилятор видит значение типа `Double`, а именно `3.5`, в контексте, где требуется `Int`. Прежде компилятор усмотрел бы в этом обычную ошибку несоответствия типов. Но теперь он сразу не сдается, а ищет средство реализации подразумеваемого преобразования `Double` в `Int`. В данном случае он находит такое средство `doubleToInt`, поскольку оно расположено в области видимости в качестве однокомпонентного идентификатора. (Вне интерпретатора `doubleToInt` можно ввести в область видимости посредством импортирования или, возможно, наследования.) Затем компилятор автоматически вставляет вызов `doubleToInt`. Закулисно код приобретает следующий вид:

```
val i: Int = doubleToInt(3.5)
```

Это действительно подразумеваемое преобразование. Оно не запрашивается в явном виде. Вместо этого `doubleToInt` помечается как доступное подразумеваемое преобразование путем ввода его в область видимости в качестве однокомпонентного идентификатора, а затем компилятор автоматически использует его при необходимости конвертирования из `Double` в `Int`.

Преобразование Double-значений в Int-значения может вызвать некоторое недоумение, поскольку сомнительна сама идея наличия чего-то, что закулисно вызывает потерю точности. То есть на самом-то деле мы не рекомендуем использовать подобное преобразование. Куда больше смысла можно усмотреть в обратном — в переходе от более ограниченного типа к более общему. Например, Int-значение может быть преобразовано без потери точности в Double-значение, следовательно, в преобразовании Int в Double есть смысл. Фактически именно так и происходит. В объекте scala.Predef, который подразумеваемо импортируется в каждую программу на Scala, определяется подразумеваемое преобразование, выполняющее конвертацию более «мелкого» типа в более «крупный». Например, в Predef можно найти следующее преобразование:

```
implicit def int2double(x: Int): Double =
x.toDouble
```

Именно поэтому в Scala Int-значения могут сохраняться в переменных типа Double. В системе типов нет для этого специальных правил, это просто подразумеваемое преобразование[119].

21.4. Преобразование получателя

Подразумеваемое преобразование применяется также к получателю вызова метода — объекту, в отношении которого вызывается метод. У этого вида подразумеваемого преобразования есть два основных варианта использования. Первый заключается в том, что преобразования получателя позволяют более гладко интегрировать новый класс в существующую иерархию классов. А второй — в поддержке возможности написания внутри базового языка предметно-ориентированных языков (domain-specific languages (DSL)).

Чтобы посмотреть все это в работе, предположим, что вы записали код `obj.doIt`, а у `obj` нет элемента по имени `doIt`. Прежде чем сдаться, компилятор попытается вставить преобразования. В данном случае преобразование следует применить к получателю `obj`. Компилятор будет действовать так, как будто у ожидаемого типа `obj` есть свойство «содержит элемент по имени `doIt`». Этот тип «содержит `doIt`» не является обычным типом Scala, но концептуально является типом, и поэтому в данном случае компилятор вставит подразумеваемое преобразование.

Взаимодействие с новыми типами

Как уже упоминалось, одно из основных применений преобразования получателя заключается в разрешении более гладкой интеграции новых типов с уже существующими типами. В частности, такие преобразования позволяют разрешить программистам клиентских программ воспользоваться экземплярами существующих типов, как будто они являются экземплярами ваших новых типов. Возьмем, к примеру, класс `Rational`, показанный в листинге 6.5. Рассмотрим еще раз фрагмент этого класса:

```
class Rational(n: Int, d: Int) {
  ...
  def + (that: Rational): Rational = ...
  def + (that: Int): Rational = ...
}
```

В классе `Rational` имеются два перегружаемых варианта метода `+`, получающих в качестве аргументов соответственно `Rational`- и `Int`-значения. Следовательно, можно сложить либо два рациональных числа, либо рациональное число с целым:

```
scala> val oneHalf = new Rational(1, 2)
oneHalf: Rational = 1/2

scala> oneHalf + oneHalf
res0: Rational = 1/1

scala> oneHalf + 1
res1: Rational = 3/2
```

А как будут обстоять дела с выражением вроде `1 + oneHalf`? Это довольно каверзное выражение, поскольку получатель `1` не располагает подходящим методом `+`. Следовательно, при выполнении следующего кода возникнет ошибка:

```
scala> 1 + oneHalf
<console>:6: error: overloaded method value + with
alternatives (Double)Double <and> ... cannot be
applied
to (Rational)
     1 + oneHalf
       ^
```

Чтобы разрешить применение подобной смешанной арифметики, нужно определить подразумеваемое преобразование из `Int` в `Rational`:

```
scala> implicit def intToRational(x: Int) =
         new Rational(x, 1)
intToRational: (x: Int)Rational
```

При использовании этого преобразования получатель проделает следующий трюк:

```
scala> 1 + oneHalf
res2: Rational = 3/2
```

Здесь компилятор Scala закулисно сначала пытается проверить соответствие типов в отношении выражения `1 + oneHalf` в его неизменном виде. Проверка проходит неудачно, поскольку в `Int` есть несколько методов `+`, но ни один из них не принимает аргумент типа `Rational`. Затем компилятор ищет подразумеваемое преобразование из `Int` в другой тип, у которого есть метод `+`, применимый к значениям типа `Rational`. Он находит ваше преобразование и применяет его, в результате чего получается следующее:

```
intToRational(1) + oneHalf
```

В данном случае компилятор находит функцию подразумеваемого преобразования, поскольку ее определение было введено в интерпретатор, который поместил это определение в область видимости для всего остального сеанса работы с интерпретатором.

Имитация нового синтаксиса

Другим основным вариантом использования подразумеваемых преобразований является имитация добавления нового синтаксиса. Вспомним, что отображение типа `Map` можно создать с помощью следующего синтаксиса:

```
Map(1 -> "one", 2 -> "two", 3 -> "three")
```

А вас не интересовало, как поддерживается `->`? Это ведь не синтаксис! Элемент вида `->` является методом класса `ArrowAssoc`, который определен внутри стандартной преамбулы Scala (`scala.Predef`). В этой преамбуле также определяется подразумеваемое преобразование из `Any` в `ArrowAssoc`. При использовании кода `1 -> "one"` компилятор вставляет преобразование из `1` в `ArrowAssoc`, чтобы мог быть найден метод `->`. Соответствующие определения выглядят таким образом:

```
package scala
object Predef {
  class ArrowAssoc[A](x: A) {
    def -> [B](y: B): Tuple2[A, B] = Tuple2(x, y)
  }
      implicit   def   any2ArrowAssoc[A](x:   A):
ArrowAssoc[A] =
    new ArrowAssoc(x)
  ...
}
```

Такая схема обогащающих оболочек часто встречается в библиотеках, предоставляющих языку расширения, похожие на синтаксис. Поэтому, как только попадется эта схема, нужно быть готовым ее распознать. Если обнаружится, что некий объект вызывает методы, не существующие в классе-получателе, скорее всего, используются подразумеваемые элементы. Аналогично, если встретится класс, имя которого означает обогащение чего-либо — `Rich`*Something* (например, `RichInt` или `RichBoolean`), то в нем, скорее всего, к типу *Something* добавляются методы, похожие на синтаксис.

Здесь эти схемы обогащающих оболочек для базовых типов, рассмотренных в главе 5, уже попадались. Теперь можно увидеть, что обогащающие оболочки применяются более широко, зачастую позволяя использовать внутренние DSL-языки, определяемые в виде библиотеки, в то время как у программистов на других языках может возникать потребность в разработке внешних DSL-языков.

Подразумеваемые классы

Подразумеваемые классы были добавлены в Scala 2.10 с целью облегчения создания обогащающих классов-оболочек. Подразумеваемым называется класс, перед определением которого ставится ключевое слово `implicit`. Для любого такого класса

компилятор создает подразумеваемое преобразование из параметра конструктора класса в сам класс. Если планируется использовать класс для схемы обогащающей оболочки, такое преобразование — именно то, что нужно.

Предположим, к примеру, что имеется класс по имени `Rectangle` для представления ширины и высоты прямоугольника на экране:

```
case class Rectangle(width: Int, height: Int)
```

Может возникнуть желание воспользоваться схемой обогащающей оболочки, чтобы упростить представление при весьма частом использовании этого класса. Один из способов решения этой задачи выглядит следующим образом:

```
implicit class RectangleMaker(width: Int) {
  def x(height: Int) = Rectangle(width, height)
}
```

В показанном ранее коде в необычной манере определяется класс с названием `RectangleMaker`. Вдобавок ко всему он вызывает автоматическое создание следующего преобразования:

```
// Создается автоматически
implicit def RectangleMaker(width: Int) =
  new RectangleMaker(width)
```

В результате можно создавать точки, помещая `x` между двумя целочисленными значениями:

```
scala> val myRectangle = 3 x 4
  myRectangle: Rectangle = Rectangle(3,4)
```

А вот как это работает: поскольку в типе `Int` нет метода по имени `x`, компилятор станет искать подразумеваемое

преобразование из `Int` во что-нибудь, имеющее этот метод. Им будет найдено созданное преобразование `RectangleMaker`, а в `RectangleMaker` имеется метод по имени `x`. Компилятор вставляет вызов этого преобразования, затем запускает в отношении `x` проверку соответствия типов и делает то, что нужно.

Особо предприимчивых следует предупредить: не стоит обольщаться насчет того, что указать ключевое слово `implicit` можно перед определением любого класса. Этот не так. В качестве подразумеваемого нельзя использовать case-класс, и у его конструктора может быть только один параметр. Кроме того, подразумеваемый класс должен размещаться внутри какого-либо другого объекта — класса или трейта. На практике, поскольку подразумеваемые классы используются в качестве обогащающих оболочек для добавления новых методов в существующий класс, эти ограничения не имеют значения.

21.5. Подразумеваемые параметры

Осталось еще одно место, куда компилятор помещает подразумеваемые элементы, — это список аргументов. Компилятор иногда заменяет код вида `someCall(a)` кодом вида `someCall(a)(b)` или код `SomeClass(a)` кодом `new SomeClass(a)(b)`, добавляя тем самым пропущенный список параметров, чтобы завершить вызов функции. Предоставляется не просто последний параметр, а весь последний каррированный список параметров. Например, если `someCall` с пропущенным списком параметров получает три параметра, компилятор может вместо `someCall(a)` вставить `someCall(a)(b, c, d)`. Чтобы воспользоваться этой возможностью, пометку `implicit` при своем определении должны иметь не только вставляемые идентификаторы, такие как `b`, `c` и `d` в `(b, c, d)`, но и последний список параметров в `someCall` или определение `someClass`.

Рассмотрим простой пример. Предположим, что имеется класс

PreferredPrompt, в котором инкапсулирована строка приглашения к вводу, используемая в оболочке (такая как, скажем, "$ " или "> ") и предпочитаемая пользователем:

```
class PreferredPrompt(val preference: String)
```

Также предположим, что имеется объект Greeter с методом greet, принимающим два списка параметров. Первый список параметров получает строку с именем пользователя, а второй — PreferredPrompt:

```
object Greeter {
     def greet(name: String)(implicit prompt:
PreferredPrompt) = {
    println("Welcome, " + name + ". The system is
ready.")
    println(prompt.preference)
  }
}
```

Последний список параметров имеет метку implicit, означающую, что он может предоставляться в подразумеваемом режиме. Но при этом по-прежнему сохраняется возможность явного указания символа приглашения к вводу с использованием следующего кода:

```
scala> val bobsPrompt = new
PreferredPrompt("relax> ")
bobsPrompt: PreferredPrompt =
PreferredPrompt@714d36d6

scala> Greeter.greet("Bob")(bobsPrompt)
Welcome, Bob. The system is ready.
relax>
```

Чтобы позволить компилятору предоставить параметр в подразумеваемом режиме, сначала нужно определить переменную ожидаемого типа, которым в данном случае будет `PreferredPrompt`. Сделать это можно, к примеру, в объекте предпочтений:

```
object JoesPrefs {
  implicit val prompt = new PreferredPrompt("Yes,
master> ")
}
```

Заметьте, что у самой `val`-переменной есть метка `implicit`. Если бы такой метки не было, компилятор не стал бы использовать эту переменную для предоставления пропущенного списка параметров. Он также не стал бы ее использовать, если бы она, как показано в следующем примере, отсутствовала в области видимости в качестве однокомпонентного идентификатора:

```
scala> Greeter.greet("Joe")
<console>:13: error: could not find implicit value
for
parameter prompt: PreferredPrompt
              Greeter.greet("Joe")
                            ^
```

Но после введения ее в область видимости посредством использования ключевого слова `import` компилятор воспользуется ею для предоставления пропущенного списка параметров:

```
scala> import JoesPrefs._
import JoesPrefs._

scala> Greeter.greet("Joe")
Welcome, Joe. The system is ready.
Yes, master>
```

Обратите внимание на то, что ключевое слово implicit применяется ко всему списку параметров, а не к отдельным параметрам. В листинге 21.1 показан пример, в котором последний параметр списка метода greet, принадлежащего объекту Greeter, который к тому же имеет метку implicit, включает два параметра: prompt типа PreferredPrompt и drink типа PreferredDrink.

Листинг 21.1. Подразумеваемый список параметров с несколькими параметрами

```
class PreferredPrompt(val preference: String)
class PreferredDrink(val preference: String)

object Greeter {
     def greet(name: String)(implicit prompt:
PreferredPrompt,
      drink: PreferredDrink) = {

    println("Welcome, " + name + ". The system is
ready.")
    print("But while you work, ")
        println("why not enjoy a cup of " +
drink.preference + "?")
    println(prompt.preference)
  }
}

object JoesPrefs {
   implicit val prompt = new PreferredPrompt("Yes,
master> ")
  implicit val drink = new PreferredDrink("tea")
```

```
}
```

В синглтон-объекте JoesPrefs объявляются две подразумеваемые val-переменные: prompt типа PreferredPrompt и drink типа PreferredDrink. Но, как и прежде, поскольку их нет в области видимости в качестве однокомпонентных идентификаторов, они не будут использоваться для заполнения пропущенного списка параметров для greet:

```
scala> Greeter.greet("Joe")
<console>:19: error: could not find implicit value
for
parameter prompt: PreferredPrompt
              Greeter.greet("Joe")
                            ^
```

Ввести обе подразумеваемые val-переменные в область видимости можно, воспользовавшись ключевым словом import:

```
scala> import JoesPrefs._
import JoesPrefs._
```

Поскольку теперь и prompt, и drink находятся в области видимости в форме однокомпонентных идентификаторов, их можно использовать для предоставления в качестве подразумеваемого списка параметров:

```
scala> Greeter.greet("Joe")(prompt, drink)
Welcome, Joe. The system is ready.
But while you work, why not enjoy a cup of tea?
Yes, master>
```

А поскольку теперь соблюдены все правила использования подразумеваемых параметров, можно задействовать альтернативный вариант и позволить компилятору Scala

предоставить вам prompt и drink, пропустив для этого указание последнего списка параметров:

```
scala> Greeter.greet("Joe")
Welcome, Joe. The system is ready.
But while you work, why not enjoy a cup of tea?
Yes, master>
```

По поводу предыдущего примера следует заметить: в нем String в качестве типа переменной prompt или переменной drink не используется, даже при том что в конечном итоге каждая из них через свои поля preference предоставляет значение типа String. Поскольку компилятор выбирает параметры путем поиска совпадения типов параметров с типом значений в области видимости, подразумеваемые параметры имеют редкие или специальные типы, для которых случайные совпадения маловероятны. Например, типы PreferredPrompt и PreferredDrink в листинге 21.1 были определены исключительно для того, чтобы послужить в качестве типов подразумеваемых параметров. Так что вряд ли эти подразумеваемые переменные данных типов находились бы в области видимости, если бы не предполагалось их использование в качестве подразумеваемых параметров для Greeter.greet.

О подразумеваемых параметрах нужно знать еще один нюанс: они, пожалуй, наиболее часто используются для предоставления информации о типе, упомянутом явным образом в более раннем списке параметров, как и классы типа языка Haskell.

Рассмотрим в качестве примера функцию maxListOrdering, показанную в листинге 21.2, которая возвращает максимальный элемент из переданного списка.

Листинг 21.2. Функция с верхним ограничителем

```
def maxListOrdering[T](elements: List[T])
      (ordering: Ordering[T]): T =
  elements match {
    case List() =>
        throw new IllegalArgumentException("empty
list!")
    case List(x) => x
    case x :: rest =>
            val maxRest = maxListOrdering(rest)
(ordering)
      if (ordering.gt(x, maxRest)) x
      else maxRest
}
```

Сигнатура `maxListOrdering` похожа на сигнатуру `orderedMergeSort`, показанную в листинге 19.12: в качестве своих аргументов функция получает `List[T]`, и теперь вводится дополнительный аргумент типа `Ordering[T]`. Этот дополнительный аргумент указывает, какой порядок следует использовать при сравнении элементов типа `T`. Таким образом, эта версия может применяться для типов, не имеющих встроенного механизма выстраивания по порядку. Кроме того, она может использоваться для типов, имеющих встроенное упорядочение, но время от времени требующих некоего другого варианта упорядочения.

Это более универсальная, но и более громоздкая в использовании версия. Теперь вызывающий объект должен указать явное упорядочение, даже если в качестве `T` применяется что-нибудь относящееся к типам `String` или `Int`, у которых явно имеются исходные механизмы упорядочения. Чтобы сделать новый метод более удобным в использовании, неплохо было бы превратить второй аргумент в подразумеваемый. Данный подход показан в листинге 21.3.

Листинг 21.3. Функция с подразумеваемым параметром

```
def maxListImpParm[T](elements: List[T])
      (implicit ordering: Ordering[T]): T =

  elements match {
    case List() =>
        throw new IllegalArgumentException("empty
list!")
    case List(x) => x
    case x :: rest =>
      val maxRest = maxListImpParm(rest)(ordering)
      if (ordering.gt(x, maxRest)) x
      else maxRest
  }
```

В примере параметр `ordering` используется для описания упорядочения значений типа `T`. В теле `maxListImpParm` это упорядочение применяется в двух местах: в рекурсивном вызове `maxListImpParm` и в выражении `if`, которое проверяет, является ли элемент `head` списка больше по размеру максимального элемента всего остального списка.

Функция `maxListImpParm` выступает примером использования подразумеваемого параметра для предоставления дополнительной информации о типе, упомянутом в явном виде в более раннем списке параметров. Если говорить точнее, подразумеваемый параметр `ordering`, относящийся к типу `Ordering[T]`, предоставляет больше информации о типе `T` — в данном случае о том, как следует выстраивать по порядку значения, относящиеся к типу `T`. Тип `T` упомянут в `List[T]` — типе элементов параметра, который появляется в более раннем списке параметров. Поскольку элементы при любом вызове `maxListImpParm` всегда должны предоставляться в явном виде, компилятор к моменту компиляции

будет знать, что такое T, и поэтому сможет определить, доступно ли подразумеваемое определение типа Ordering[T]. Если оно окажется доступно, компилятор сможет передать второй список параметров ordering, воспользовавшись механизмом предоставления подразумеваемых параметров.

Эта схема получила настолько широкое распространение, что стандартная библиотека Scala предоставляет подразумеваемые методы упорядочения для множества самых востребованных типов. Благодаря этому методом maxListImpParm можно воспользоваться при работе с различными типами:

```
scala> maxListImpParm(List(1,5,10,3))
res9: Int = 10

scala> maxListImpParm(List(1.5, 5.2, 10.7, 3.14159))
res10: Double = 10.7

scala> maxListImpParm(List("one", "two", "three"))
res11: String = two
```

В первом случае компилятор вставит упорядочение для Int-значений, во втором — для Double-значений, а в третьем — для String-значений.

Стилевое правило для подразумеваемых параметров. Согласно стилевому правилу для типов подразумеваемых параметров лучше воспользоваться типом, названным особым образом. Например, переменные prompt и drink в предыдущем примере относились не к типу String, а к типам PreferredPrompt и PreferredDrink соответственно. В качестве контрпримера рассмотрим возможность того, что функция maxListImpParm может быть записана с использованием следующей сигнатуры типа:

```
def maxListPoorStyle[T](elements: List[T])
      (implicit orderer: (T, T) => Boolean): T
```

Но, чтобы воспользоваться этой версией функции, вызывающему объекту придется предоставить параметр `orderer`, относящийся к типу `(T, T) => Boolean`. Это довольно-таки общий тип, включающий любую функцию, выдающую на основе двух значений типа `T` результат типа `Boolean`. Из него абсолютно не ясно предназначение этого типа — это может быть проверка на равенство, на то, что одно значение больше или меньше другого, или же что-то иное.

В настоящем коде для `maxListImpParm`, который приведен в листинге 21.3, показан более удачный стиль. В нем используется параметр `ordering`, относящийся к типу `Ordering[T]`. Слово `Ordering` в наименовании типа явно указывает на цель использования подразумеваемого параметра: он предназначен для упорядочения элементов `T`-типа. Поскольку этот тип упорядочения указан явно, с добавлением в стандартную библиотеку подразумеваемых представителей этого типа не возникнет никаких проблем. Но представьте себе хаос, который возникнет при добавлении в стандартную библиотеку подразумеваемого элемента типа `(T, T) => Boolean`, когда компилятор начнет разбрасывать его в чьем-то коде. В результате будет получен код, который пройдет компиляцию и сможет запускаться на выполнение, но проверки в отношении пар элементов будут выполняться как попало! Стало быть, нужно придерживаться следующего стилевого правила: использовать в типе подразумеваемого параметра хотя бы одно имя, определяющее его роль.

21.6. Контекстные ограничители

В предыдущем примере возможность применения подразумеваемого элемента была показана, но не реализована.

Следует заметить, что при использовании подразумеваемого элемента в качестве параметра компилятор не только попытается предоставить этот параметр с подразумеваемым значением, но и воспользуется им в качестве доступного подразумеваемого элемента в теле метода! Таким образом, первое использование `ordering` внутри тела метода может быть опущено.

Когда компилятор станет анализировать код, показанный в листинге 21.4, он увидит, что типы не совпадают. Выражению `maxList(rest)` предоставлен только один список параметров, а `maxList` требует два списка. Поскольку второй список параметров является подразумеваемым, компилятор не спешит сдаваться на проверке соответствия типов. Вместо этого он ищет подразумеваемый параметр подходящего типа, в данном случае `Ordering[T]`. Что касается нашего кода, то компилятор находит такой параметр и перезаписывает вызов в `maxList(rest)` `(ordering)`, после чего код успешно проходит проверку соответствия типов.

Листинг 21.4. Функция, использующая подразумеваемый параметр внутри себя

```
def maxList[T](elements: List[T])
      (implicit ordering: Ordering[T]): T =

  elements match {
    case List() =>
        throw new IllegalArgumentException("empty
list!")
    case List(x) => x
    case x :: rest =>
        val maxRest = maxList(rest)       // здесь
(ordering) подразумевается
        if (ordering.gt(x, maxRest)) x   // а этот
```

```
ordering по-прежнему
       else maxRest                              // указан
явно
  }
```

Существует также способ исключения второго использования `ordering`. Для него применяется следующий метод, определение которого находится в стандартной библиотеке:

```
def implicitly[T](implicit t: T) = t
```

Эффект от вызова `implicitly[Foo]` заключается в том, что компилятор станет искать подразумеваемое определение типа `Foo`. Затем он вызывает в отношении этого объекта подразумеваемый метод, который в свою очередь возвращает объект обратно. Таким образом, можно воспользоваться кодом `implicitly[Foo]` везде, где нужно найти подразумеваемый объект типа `Foo` в текущей области видимости. Например, в листинге 21.5 показано использование `implicitly[Ordering[T]]` для извлечения параметра `ordering` по его типу.

Листинг 21.5. Функция, используемая подразумеваемым образом

```
def maxList[T](elements: List[T])
       (implicit comparator: Ordering[T]): T = //
то же самое тело...

  def maxList[T](elements: List[T])
       (implicit ordering: Ordering[T]): T =

    elements match {
      case List() =>
         throw new IllegalArgumentException("empty
```

```
list!")
      case List(x) => x
      case x :: rest =>
        val maxRest = maxList(rest)
                if  (implicitly[Ordering[T]].gt(x,
maxRest)) x
        else maxRest
    }
```

Присмотритесь к последней версии `maxList` повнимательнее. В ней в тексте метода нет ни одного упоминания о параметре `ordering`. Второй параметр можно также назвать `comparator`.

По той же причине работает и эта версия:

```
def maxList[T](elements: List[T])
       (implicit iceCream: Ordering[T]): T = // то
же самое тело ...
```

Поскольку данная схема получила широкое распространение, в Scala допускается не указывать имя этого параметра и сократить заголовок метода, используя *контекстные ограничители*. В этом случае можно будет записывать сигнатуру `maxList`, как показано в листинге 21.6. Синтаксис `[T : Ordering]` является контекстным ограничителем с двойным предназначением. Во-первых, он вводит в качестве обычного параметр типа `T`. Во-вторых, добавляет подразумеваемый параметр типа `Ordering[T]`. В предыдущих версиях `maxList` этот параметр назывался `ordering`, но при использовании контекстного ограничителя неизвестно, какой именно параметр будет вызываться. Как было показано ранее, зачастую знать, какой именно параметр вызывается, и не нужно.

Листинг 21.6. Функция с контекстным ограничителем

```
def maxList[T : Ordering](elements: List[T]): T =
```

```
  elements match {
    case List() =>
        throw new IllegalArgumentException("empty
list!")
    case List(x) => x
    case x :: rest =>
      val maxRest = maxList(rest)
       if (implicitly[Ordering[T]].gt(x, maxRest))
x
      else maxRest
  }
```

На интуитивном уровне контекстный ограничитель можно воспринимать как высказывание о параметре типа. При записи `[T <: Ordered[T]]` говорится, что `T` является типом `Ordered[T]`. В отличие от этого, при записи `[T : Ordering]` о том, что такое `T`, столь подробно не сообщается — говорится о наличии некой формы упорядочения, связанной с `T`. Таким образом, контекстный ограничитель обладает достаточной гибкостью. Он позволяет использовать код, требующий упорядочения или любого другого свойства типа без необходимости изменять определение этого типа.

21.7. Когда применяются множественные преобразования

Может случиться так, что в области видимости будут находиться сразу несколько подразумеваемых преобразований, каждое из которых должно работать. В большинстве случаев Scala в такой ситуации отказывается вставлять преобразование. Подразумеваемые элементы хорошо работают, когда преобразование остается абсолютно очевидным и схематически чистым. Если применяется сразу несколько преобразований, то выбор не столь очевиден.

Рассмотрим простой пример. В нем есть метод, получающий последовательность, преобразование, превращающее целочисленное значение в диапазон, и преобразование, превращающее целочисленное значение в список цифр:

```
scala> def printLength(seq: Seq[Int]) = println(seq.length)
printLength: (seq: Seq[Int])Unit

scala> implicit def intToRange(i: Int) = 1 to i
intToRange: (i: Int)scala.collection.immutable.Range.Inclusive

scala> implicit def intToDigits(i: Int) =
         i.toString.toList.map(_.toInt)
intToDigits: (i: Int)List[Int]

scala> printLength(12)
<console>:26: error: type mismatch;
found : Int(12)
required: Seq[Int]
Note that implicit conversions are not applicable because
they are ambiguous:
both method intToRange of type (i: Int)scala.collection.immutable.Range.Inclusive
and method intToDigits of type (i: Int)List[Int]
are possible conversion functions from Int(12) to Seq[Int]
              printLength(12)
                          ^
```

Здесь, несомненно, имеется неоднозначность. Преобразование

целочисленного значения в цифры коренным образом отличается от его преобразования в диапазон. В данном случае программист должен указать, какое из преобразований предназначено для того, чтобы стать подразумеваемым. Вплоть до выхода Scala 2.7 на этом вся история и заканчивалась. При применении сразу нескольких подразумеваемых преобразований компилятор отказывался делать выбор между ними. Возникала такая же ситуация, как и с перегрузкой методов. При попытке вызова `foo(null)` и наличии двух различных перегружаемых методов `foo`, допускающих `null`, компилятор отвергал код. Он говорил, что цель вызова метода неоднозначна.

В Scala 2.8 это правило было смягчено. Если одно из доступных преобразований конкретнее других, компилятор выберет его. Замысел состоит в том, что при наличии достаточных оснований полагать, что программист всегда выбрал бы только одно из имеющихся преобразований, программисту не требуется записывать это преобразование в явном виде. Ведь у правила перегрузки методов имеется точно такое же послабление. Если продолжить рассмотрение предыдущего примера, то при условии, что один из доступных методов `foo` получает значение типа `String`, а другие получают значение типа `Any`, выбор будет за версией, получающей `String`. Вполне очевидно, что этот вариант более конкретен.

Уточним сказанное. Одно подразумеваемое преобразование будет конкретнее другого, если к нему применимо какое-либо из следующих утверждений.

- Тип аргумента первого преобразования является подтипом второго преобразования.
- Оба преобразования являются методами, и класс, в который входит первое, расширяет класс, в который входит второе.

Мотивом для возвращения к этому вопросу и пересмотра

правила было улучшение взаимодействия между коллекциями Java, коллекциями Scala и строками.

Рассмотрим простой пример:

```
val cba = "abc".reverse
```

Какой тип будет выведен для `cba`? Интуитивно понятно, что тип должен быть `String`. Реверсирование строки будет порождать другую строку, не так ли? Но в Scala 2.7 получалось так, что строка `"abc"` превращалась в коллекцию Scala. Реверсирование коллекции Scala порождало коллекцию Scala, следовательно, типом `cba` будет коллекция. Там также происходило подразумеваемое преобразование обратно в строку, но это не решало все проблемы. К примеру, в версии, предшествующей Scala 2.8, выражение `"abc" == "abc".reverse.reverse` вычислялось в `false`!

При использовании Scala 2.8 типом `cba` является `String`. Старое подразумеваемое преобразование в коллекцию Scala (которое теперь называется `WrappedString`) сохраняется. Но предоставляет более конкретное преобразование из `String` в новый тип под названием `StringOps`. В `StringOps` имеется множество таких методов, как `reverse`, но вместо возвращения коллекции они возвращают значение типа `String`. Преобразование в `StringOps` задано непосредственно в `Predef`, а преобразование в коллекцию Scala — в новом классе `LowPriorityImplicits`, который расширяется за счет `Predef`. Там, где есть выбор из этих двух преобразований, компилятор выбирает преобразование в `StringOps`, поскольку оно определено в подклассе того класса, где задано другое преобразование.

21.8. Отладка подразумеваемых элементов

Подразумеваемые элементы являются довольно мощной функцией языка Scala, но временами с ней трудно справиться должным

образом. В этом разделе содержится ряд советов по отладке кода, использующего подразумеваемые элементы.

Иногда можно удивиться, почему компилятор не находит подразумеваемого преобразования, которое, на ваш взгляд, он должен применить. В таком случае помогает явное указание преобразования. Если компилятор к тому же выдает сообщение об ошибке, то причина, по которой компилятор не может применить ваше подразумеваемое преобразование, известна.

Предположим, к примеру, что `wrapString` была ошибочно выбрана для преобразования из `String`- в `List`-значения, а не в `IndexedSeq`-значения. Тогда может возникнуть вопрос, почему не работает следующий код:

```
scala> val chars: List[Char] = "xyz"
<console>:24: error: type mismatch;
found   : String("xyz")
required: List[Char]
       val chars: List[Char] = "xyz"
                               ^
```

И опять, чтобы понять, что именно пошло не так, поможет явное указание преобразования `wrapString`:

```
scala> val chars: List[Char] = wrapString("xyz")
<console>:24: error: type mismatch;
found : scala.collection.immutable.WrappedString
required: List[Char]
       val chars: List[Char] = wrapString("xyz")
                                          ^
```

Его использование позволит выяснить причину ошибки: у `wrapString` неверный тип возвращаемого значения. Возможно также, что вставка явного указания на преобразование приведет к устранению ошибки. В этом случае становится известно, что

применению подразумеваемого преобразования помешало нарушение одного из дополнительных правил (правила области видимости).

Иногда при отладке программы поможет взгляд на то, какое именно из подразумеваемых преобразований было вставлено компилятором. Для этого пригодится ключ запуска компилятора -Xprint:typer. При запуске scalac с этим ключом компилятор покажет, как выглядит код после добавления механизмов проверки соответствия типов всех подразумеваемых преобразований. Пример приведен в листингах 21.7 и 21.8. Если посмотреть на последнюю инструкцию каждого из этих листингов, можно увидеть, что второй список параметров для enjoy, который был пропущен в листинге 21.7, enjoy("reader"), был вставлен компилятором, что и показано в листинге 21.8:

```
Mocha.this.enjoy("reader")(Mocha.this.pref)
```

Листинг 21.7. Пример кода, использующего подразумеваемый параметр

```
object Mocha extends App {

  class PreferredDrink(val preference: String)

  implicit val pref = new PreferredDrink("mocha")

      def enjoy(name: String)(implicit drink:
PreferredDrink) = {
    print("Welcome, " + name)
    print(". Enjoy a ")
    print(drink.preference)
    println("!")
  }
```

```
  enjoy("reader")
}
```

Листинг 21.8. Пример кода после проверки соответствия типов и вставки подразумеваемых элементов

```
$ scalac -Xprint:typer mocha.scala
[[syntax trees at end of typer]]
// Исходный код на Scala source: mocha.scala
package <empty> {
      final object Mocha extends java.lang.Object
with Application
        with ScalaObject {
      // ...

       private[this] val pref: Mocha.PreferredDrink
=
        new Mocha.this.PreferredDrink("mocha");
      implicit <stable> <accessor>
                def pref: Mocha.PreferredDrink =
Mocha.this.pref;
      def enjoy(name: String)
            (implicit drink: Mocha.PreferredDrink):
Unit = {
              scala.this.Predef.print("Welcome, ".+
(name));
        scala.this.Predef.print(". Enjoy a ");
        scala.this.Predef.print(drink.preference);
        scala.this.Predef.println("!")
      };
      Mocha.this.enjoy("reader")(Mocha.this.pref)
```

```
    }
  }
```

Можете отважиться и на попытку использования команды `scala -Xprint:typer`, чтобы получить интерактивную оболочку, выводящую исходный код, используемый внутри интерпретатора, после набора текста.

Резюме

Подразумеваемые элементы являются довольно мощной функцией Scala, существенно сокращающей объем исходного кода. В этой главе были рассмотрены действующие в Scala правила, касающиеся подразумеваемых элементов, и ряд наиболее распространенных в программировании ситуаций, когда, используя подразумеваемые элементы, можно получить весомые преимущества.

В качестве предостережения следует заметить, что подразумеваемые элементы при слишком частом использовании могут запутать код. Поэтому, прежде чем добавлять новое подразумеваемое преобразование, задайтесь вопросом: нельзя ли получить аналогичный эффект с применением других средств, таких как наследование, составление подмешиваний или перегрузка методов? Если ни одно из этих средств не подойдет и у вас возникнет ощущение, что в вашем коде все еще остается много громоздкого и избыточного, тогда улучшить ситуацию сможет применение подразумеваемых элементов.

117 В разделе 31.5 мы объясним, что он будет работать в Scala 2.12.

118 В качестве подразумеваемых параметров могут использоваться переменные и синглтон-объекты с пометкой implicit. Этот вариант будет рассмотрен в данной главе чуть позже.

119 Но внутренний код компилятора Scala будет рассматривать преобразование особым образом, переводя его в специальный байт-код i2d. Следовательно, скомпилированный образ будет таким же, как и в Java.

22. Реализация списков

Списки в данной книге встречались повсеместно. Класс `List`, наверное, является одним из самых востребованных в Scala типов структурированных данных. Использование списков было показано в главе 16. А в этой главе срываются покровы и объясняется сама суть реализации списков в Scala.

Разбираться во внутренней работе класса `List` полезно по нескольким причинам. Так вы приобретете более четкое представление об относительной эффективности операций со списками, помогающее быстро создавать компактный код, использующий списки. Освоите набор методов, который можно применить при разработке собственных библиотек. И наконец, класс `List` — это пример искусного применения имеющихся в Scala системы типов в целом и концепции универсального типа в частности. Следовательно, изучение класса `List` позволит углубить ваши знания в этих областях.

22.1. Принципиальный взгляд на класс List

Списки не являются в Scala встроенной конструкцией — они определяются с применением абстрактного класса `List` из пакета `scala`, поставляемого с двумя подклассами для `::` и `Nil`. В этой главе будет представлен беглый обзор класса `List`. В данном разделе приводится несколько упрощенная версия класса по сравнению с его реальной реализацией в стандартной библиотеке Scala, рассматриваемой в разделе 22.3:

```
package scala
abstract class List[+T] {
```

`List` является абстрактным классом, следовательно, определить элементы путем вызова пустого конструктора `List`

невозможно. Например, выражение `new List` будет запрещенным. У класса имеется параметр типа `T`. Знак `+` перед указанием этого параметра типа является признаком ковариантности списков, рассмотренным в главе 19.

Благодаря этому свойству переменной типа `List[Any]` можно присвоить значение типа `List[Int]`:

```
scala> val xs = List(1, 2, 3)
xs: List[Int] = List(1, 2, 3)

scala> var ys: List[Any] = xs
ys: List[Any] = List(1, 2, 3)
```

Все операции со списками могут быть определены в понятиях трех основных методов:

```
def isEmpty: Boolean
def head: T
def tail: List[T]
```

Все эти методы являются абстрактными, принадлежащими классу `List`. Они определены в подчиненном объекте `Nil` и в подклассе `::`. Иерархия класса `List` показана на рис. 22.1.

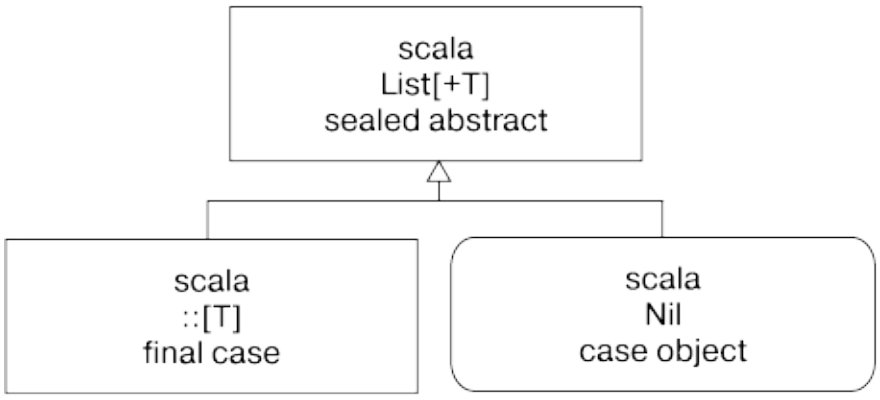


Рис. 22.1. Иерархия классов для списков Scala

Объект Nil

Объект `Nil` определяется как пустой список, что показано в листинге 22.1. Этот объект является наследником типа `List[Nothing]`. Благодаря ковариантности это означает, что `Nil` совместим с каждым экземпляром типа `List`.

Листинг 22.1. Определение синглтон-объекта Nil

```
case object Nil extends List[Nothing] {
  override def isEmpty = true
  def head: Nothing =
       throw new NoSuchElementException("head of
empty list")
  def tail: List[Nothing] =
       throw new NoSuchElementException("tail of
empty list")
}
```

В объекте `Nil` очень просто реализуются три абстрактных метода класса `List`: метод `isEmpty` возвращает `true`, а оба метода, `head` и `tail`, выдают исключения. Следует заметить, что выдача исключения оказывается не только обоснованным, но и единственно возможным действием для метода `head`: поскольку `Nil` является списком `List` из ничего, `Nothing`, типом результата `head` должен быть `Nothing`. А так как значения такого типа не существует, это означает, что метод `head` не может вернуть обычное значение. Ему приходится выполнять возвращение ненормальным образом — через выдачу исключения[120].

Класс ::

Класс `::` (произносится «конс», от слова construct — «создавать») представляет непустые списки. Он назван так с целью поддержки

поиска по образцу с инфиксом `::`. В разделе 16.5 было показано, что каждая инфиксная операция в образце рассматривается как применение конструктора инфиксного оператора к его аргументам. Следовательно, схема `x :: xs` выступает как `::(x, xs)`, где `::` является case-классом.

Определение case-класса `::` выглядит следующим образом:

```
final case class ::[T](hd: T, tl: List[T]) extends
List[T] {
  def head = hd
  def tail = tl
  override def isEmpty: Boolean = false
}
```

Реализация класса `::` весьма проста. Он получает два конструируемых параметра, `hd` и `tl`, представляющих голову и хвост создаваемого списка.

Определения методов `head` и `tail` просто возвращают соответствующий параметр. Фактически эта схема может быть сокращена за счет разрешения параметрам непосредственно реализовывать методы `head` и `tail` родительского класса `List`, как в следующем эквивалентном, но более лаконичном определении класса `::`:

```
final case class ::[T](head: T, tail: List[T])
    extends List[T] {
  override def isEmpty: Boolean = false
}
```

Этот код работает благодаря тому, что каждый параметр case-класса является также и полем класса (это похоже на объявление параметра с префиксом `val`). Вспомним, что в разделе 20.3 говорилось, что Scala позволяет выполнять реализацию абстрактных методов, не имеющих параметров, таких как `head`

или tail, с полем. Следовательно, показанный ранее код напрямую использует параметры head и tail в качестве реализаций абстрактных методов head и tail, унаследованных из класса List.

Еще несколько методов

Все остальные методы класса List могут быть написаны с использованием трех основных методов, например:

```
def length: Int =
  if (isEmpty) 0 else 1 + tail.length
```

или:

```
def drop(n: Int): List[T] =
  if (isEmpty) Nil
  else if (n <= 0) this
  else tail.drop(n - 1)
```

или:

```
def map[U](f: T => U): List[U] =
  if (isEmpty) Nil
  else f(head) :: tail.map(f)
```

Создание списка

Методы создания списка :: и ::: имеют особое свойство. Поскольку их имена заканчиваются двоеточием, они привязаны к своему правому операнду. То есть такая операция, как x :: xs, рассматривается в качестве вызова метода xs.::(x), а не как вызов метода x.::(xs). Фактически выражение x.::(xs) не имело бы никакого смысла, поскольку x относится к типу элементов списка, который может быть каким угодно, поэтому

предположить, что в этом типе будет реализован метод `::`, невозможно.

Таким образом, метод `::` должен получить значение элемента и выдать новый список. Каков же обязательный тип значения элемента? Возможно, вам захочется сказать в ответ, что он должен быть таким же, как и тип элементов списка, но на самом деле он имеет более ограниченную природу, чем нужно.

Чтобы понять причину этого, рассмотрим иерархию следующего класса:

```
abstract class Fruit
class Apple extends Fruit
class Orange extends Fruit
```

В листинге 22.2 показано, что произойдет при создании списка фруктов (`fruits`).

Листинг 22.2. Добавление элемента родительского типа в список подтипов

```
scala> val apples = new Apple :: Nil
apples: List[Apple] = List(Apple@e885c6a)

scala> val fruits = new Orange :: apples
fruits: List[Fruit] = List(Orange@3f51b349,
Apple@e885c6a)
```

Значение `apples` (яблоки) согласно ожиданиям рассматривается как список, состоящий из элементов типа `Apple`. Но определение фруктов показывает, что сохраняется возможность добавления к этому списку элемента другого типа. Типом элемента получающегося списка объявляется `Fruit`, являющийся наиболее точным общим родительским типом для типа элементов исходного

списка, то есть Apple, и типа добавляемого элемента, то есть Orange. Эта гибкость достигается за счет определения метода :: (cons), показанного в листинге 22.3.

Листинг 22.3. Определение метода :: (cons) в классе List

```
def ::[U >: T](x: U): List[U] = new scala.::(x,
this)
```

Следует заметить, что сам по себе этот метод является полиморфным, поскольку он получает параметр типа по имени U. Кроме того, на U наложены ограничения в выражении [U >: T], и он должен быть родительским типом того типа T, к которому относятся элементы списка. Требуется, чтобы добавляемый элемент относился к типу U, а получаемый результат — к типу List[U].

С той формулировкой ::, которая дана в листинге 22.3, можно проверить, как определение fruits, показанное в листинге 22.2, срабатывает в зависимости от типа: в этом определении параметр типа U класса :: конкретизируется в Fruit. Условие по нижнему ограничителю U удовлетворено, поскольку список apples относится к типу List[Apple], а Fruit является родительским типом для типа Apple. Аргументом метода :: является new Orange, который соответствует типу Fruit. Поэтому применение метода не приводит к нарушению соответствия типов, а типом его результата оказывается List[Fruit]. На рис. 22.2 показана структура списков, получающихся в результате выполнения кода из листинга 22.2.

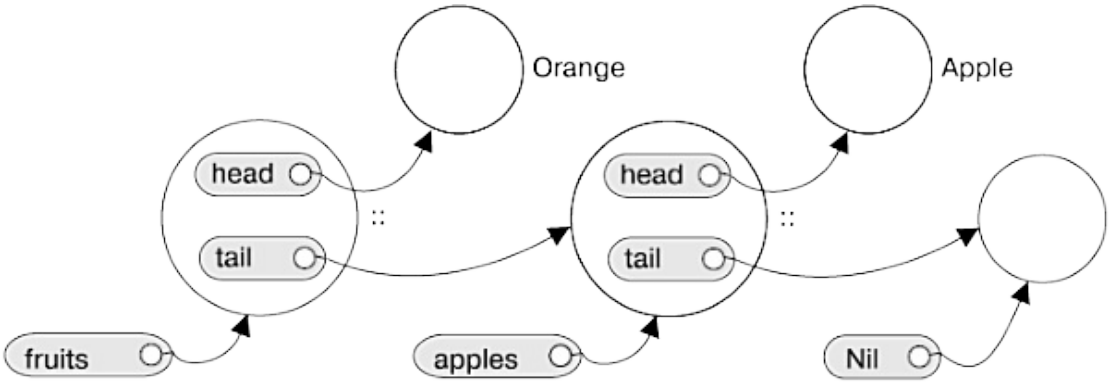


Рис. 22.2. Структура списков Scala, показанных в листинге 22.2

Фактически полиморфное определение метода `::` с нижним ограничителем `T` не только приносит определенные удобства, но даже необходимо для представления определения класса `List` в корректном в отношении типов виде. Это происходит благодаря тому, что списки типа `List` определены как ковариантные.

Предположим на минуту, что у метода `::` имеется следующее определение:

```
// Мысленный эксперимент (в неработоспособном виде)
def ::(x: T): List[T] = new scala.::(x, this)
```

В главе 19 было показано, что параметры этого метода считаются контрвариантными позициями, следовательно, элементы списка типа `T` находятся в показанном ранее определении в контрвариантной позиции. Но тогда класс `List` в `T` не может быть объявлен ковариантным. Таким образом, нижний ограничитель `[U >: T]` убивает одним выстрелом двух зайцев: устраняет проблему несоответствия типов и приводит к повышению гибкости использования метода `::`. Как показано в листинге 22.4, метод объединения списков `:::` определен по аналогии с методом `::`.

Листинг 22.4. Определение метода ::: в классе List

```
def :::[U >: T](prefix: List[U]): List[U] =
if (prefix.isEmpty) this
else prefix.head :: prefix.tail ::: this
```

Как и метод `cons`, метод объединения является полиморфным. Тип результата расширяется по мере необходимости для включения типов всех элементов списка. Нужно еще раз отметить, что порядок следования аргументов между инфиксной операцией и явным вызовом метода меняется на противоположный. Поскольку имена методов `:::` и `::` заканчиваются двоеточием, оба они привязаны к правому аргументу и оба обладают правой ассоциативностью. Например, в части `else` определения метода `:::`, показанного в листинге 22.4, содержатся инфиксные операции с использованием как оператора `::`, так и оператора `:::`.

Эти инфиксные операции могут быть расширены в следующие эквивалентные вызовы методов:

```
prefix.head :: prefix.tail ::: this
```

является эквивалентом (поскольку `::` и `:::` обладают правой ассоциативностью):

```
prefix.head :: (prefix.tail ::: this)
```

является эквивалентом (поскольку `::` привязывается к правому операнду):

```
(prefix.tail ::: this).::(prefix.head)
```

является эквивалентом (поскольку `:::` привязывается к правому операнду):

```
this.:::(prefix.tail).::(prefix.head)
```

22.2. Класс ListBuffer

Обычная схема доступа к списку имеет рекурсивный характер. Например, чтобы увеличить на единицу значение каждого элемента списка без использования метода `map`, можно воспользоваться следующим кодом:

```
def incAll(xs: List[Int]): List[Int] = xs match {
  case List() => List()
  case x :: xs1 => x + 1 :: incAll(xs1)
}
```

Одним из недостатков этой программной схемы является то, что она не является концевой рекурсией. Следует заметить, что показанный ранее рекурсивный вызов `incAll` происходит внутри операции `::`. Поэтому каждый рекурсивный вызов требует нового стекового фрейма.

На современных виртуальных машинах это означает, что применять `incAll` к спискам, количество элементов которых существенно превышает диапазон от 30 000 до 50 000, невозможно. А жаль. А как бы вы написали версию `incAll`, способную работать со списками произвольного размера (насколько это позволил бы объем динамической области памяти)?

Одним из подходов может стать применение цикла

```
for (x <- xs) // ??
```

Но что должно быть помещено в тело цикла? Ведь `incAll` создает список, помещая элементы перед результатом рекурсивного вызова, а циклу нужно добавлять новые элементы к концу получающегося в результате списка. Возможность использования оператора добавления списка `:::` представляется слишком неэффективной:

```
var result = List[Int]() // весьма неэффективный
подход
for (x <- xs) result = result ::: List(x + 1)
```

```
result
```

Это абсолютно неэффективный код. Поскольку на выполнение операции `:::` затрачивается время, пропорциональное длине ее первого операнда, на всю операцию уйдет время, пропорциональное квадрату длины списка. А это совершенно неприемлемо.

Более удачной альтернативой будет использование списочного буфера, позволяющего аккумулировать элементы списка. Для этого нужно воспользоваться такой операцией, как `buf += elem`, добавляющей элемент `elem` к концу списочного буфера `buf`. После завершения добавления элементов списочный буфер можно превратить в обычный список, воспользовавшись операцией `toList`.

`ListBuffer` является классом, который находится в пакете `scala.collection.mutable`. Чтобы пользоваться только простым именем, можно импортировать `ListBuffer` из его пакета:

```
import scala.collection.mutable.ListBuffer
```

Теперь с использованием списочного буфера тело `incAll` можно записать следующим образом:

```
val buf = new ListBuffer[Int]
for (x <- xs) buf += x + 1
buf.toList
```

Это весьма эффективный способ создания списков. Фактически реализация списочного буфера организована таким образом, чтобы на обе операции, добавления (`+=`) и превращения в список (`toList`), уходило постоянное, весьма незначительное время.

22.3. Практическое использование класса List

Реализация методов работы со списками, рассмотренная в разделе 22.1, имеет весьма лаконичный и понятный код, но страдает от проблемы переполнения так же, как и реализация `incAll` без концевой рекурсии. Поэтому в большинстве методов в реальной реализации класса `List` используется не рекурсия, а циклы со списочными буферами. Например, в листинге 22.5 показана реальная реализация метода `map` в классе `List`.

Листинг 22.5. Определение метода map в классе List

```
final override def map[U](f: T => U): List[U] = {
  val b = new ListBuffer[U]
  var these = this
  while (!these.isEmpty) {
    b += f(these.head)
    these = these.tail
  }
  b.toList
}
```

Эта пересмотренная реализация выполняет проход по элементам списка с применением простого, но эффективного цикла. Не менее эффективной будет и реализация с использованием концевой рекурсии, но общая рекурсивная реализация будет работать медленнее и хуже поддаваться масштабированию. А как охарактеризовать операцию `b.toList` в конце кода? Какова ее вычислительная сложность? Фактически вызов метода `toList` занимает весьма незначительное количество циклов, не зависящее от длины списка.

Чтобы понять причину этого, давайте еще раз взглянем на класс `::`, создающий непустые списки. На практике этот класс не вполне соответствует своему идеализированному определению, которое было дано в разделе 22.1. Реальное определение показано в

листинге 22.6. Как видите, в нем есть еще одна особенность: аргумент `tl` является `var`-переменной! Это означает, что хвост после создания списка можно изменять. Но, поскольку у переменной `tl` имеется модификатор `private[scala]`, к ней можно обращаться только из пакета `scala`. Клиентский код, находящийся за пределами этого пакета, не сможет выполнять чтение переменной `tl` и запись в нее.

Листинг 22.6. Определение подкласса :: класса List

```
final case class ::[U](hd: U,
        private[scala] var tl: List[U]) extends
List[U] {
  def head = hd
  def tail = tl
  override def isEmpty: Boolean = false
}
```

Поскольку класс `ListBuffer` содержится в подпакете `scala.collection.mutable` пакета `scala`, `ListBuffer` может обращаться к полю `tl` ячейки `cons`. Фактически элементы списочного буфера представлены в виде списка, и добавление новых элементов включает изменение поля `tl` последней ячейки `::` в этом списке. Начало кода класса `ListBuffer` выглядит следующим образом:

```
package scala.collection.immutable
final class ListBuffer[T] extends Buffer[T] {
  private var start: List[T] = Nil
  private var last0: ::[T] = _
  private var exported: Boolean = false
  ...
```

Здесь видны три закрытых поля, характеризующих `ListBuffer`:

- `start` указывает на список всех элементов, хранящихся в списочном буфере;
- `last0` указывает на последнюю ячейку `::` в этом списочном буфере;
- `exported` показывает, превращен ли с `toList` списочный буфер в список.

Операция `toList` очень проста:

```
override def toList: List[T] = {
  exported = !start.isEmpty
  start
}
```

Она возвращает список элементов, на который ссылается `start`, а также присваивает переменной `exported` значение `true`, если список не пуст. Следовательно, операция `toList` весьма эффективна, поскольку она не копирует список, сохраненный в `ListBuffer`.

А что произойдет, если список после выполнения операции `toList` будет дополнен еще раз? Разумеется, если список возвращен из `toList`, он должен быть неизменяемым. Но добавление к элементу `last0` приведет к изменению списка, на который ссылается `start`.

Для поддержки корректности операций со списочным буфером нужно вместо этого работать со свежим списком. Такая задача в реализации операции `+=` решается в первой же строке:

```
override def += (x: T) = {
```

```
    if (exported) copy()
    if (start.isEmpty) {
      last0 = new scala.::(x, Nil)
      start = last0
    } else {
      val last1 = last0
      last0 = new scala.::(x, Nil)
      last1.tl = last0
    }
  }
```

Как видите, операция `+=` копирует список, на который указывает `start`, если переменная `exported` имеет значение `true`. Следовательно, в итоге без издержек не обходится. Если понадобится перейти от списков, допускающих расширение, к неизменяемым спискам, без копирования ничего не получится. Но реализация `ListBuffer` выполнена таким образом, что копирование необходимо только для списочных буферов, расширение которых продолжилось после их превращения в списки.

На практике такое случается очень редко. В большинстве случаев использования списочных буферов элементы добавляются постепенно, а затем в конце выполняется всего одна операция `toList`. Тогда в копировании нет необходимости.

22.4. Внешняя функциональность

В предыдущем разделе были рассмотрены основные элементы реализации в Scala классов `List` и `ListBuffer`. Было показано, что снаружи списки имеют чисто функциональную природу, но внутри у них есть императивная реализация, использующая списочные буферы. Это обычная стратегия в программировании на Scala, заключающаяся в стремлении добиться сочетания чистоты с

эффективностью путем тщательного ограничения последствий нечистых операций.

У вас может возникнуть вопрос: зачем так рьяно настаивать на чистоте? Почему бы просто не открыть определение списков, сделав изменяемым поле `tail`, а может быть, и поле `head`? Недостаток такого подхода заключается в том, что программы станут существенно менее надежными. Следует заметить, что при создании списков с помощью оператора `::` многократно используется хвост создаваемого списка.

Поэтому при написании кода

```
val ys = 1 :: xs
val zs = 2 :: xs
```

хвосты списков `ys` и `zs` являются общими — они указывают на одну и ту же структуру данных. Это важно с точки зрения эффективности. Если бы список `xs` копировался всякий раз при добавлении к нему нового элемента, то работа выполнялась бы намного медленнее. Поскольку совместное использование получило весьма широкое распространение, изменение элементов списка, если бы оно было возможно, было бы крайне опасной операцией.

К примеру, если путем написания следующего кода в показанном ранее фрагменте понадобится выполнить усечение списка `ys` до его первых двух элементов:

```
ys.drop(2).tail = Nil // сделать это в Scala нельзя!
```

то в качестве побочного эффекта будут усечены также списки `zs` и `xs`.

Совершенно очевидно, что отслеживать места изменений крайне трудно. Именно поэтому для списков в Scala выбраны их широкомасштабное совместное применение и отказ от изменений.

Если пожелаете, то класс `ListBuffer` по-прежнему позволяет создавать списки постепенно, в императивном стиле. Но, поскольку списочные буферы не относятся к обычным спискам, в типах изменяемые списочные буферы и неизменяемые списки содержатся раздельно.

В конструкциях имеющихся в Scala классов `List` и `ListBuffer` происходят действия, очень похожие на те, что выполняются в принадлежащей Java паре классов `String` и `StringBuffer`. Это не случайно. В обеих ситуациях разработчики стремились создать чистые неизменяемые структуры данных, но при этом представить эффективный способ постепенного создания таких структур. Для строк Java и Scala методы `StringBuffer` (или в Java 5 — `StringBuilder`) предоставляют способ постепенного создания строки. В списках Scala у вас есть выбор: можно либо создавать списки постепенно путем добавления элементов к началу списка с помощью операции `::`, либо воспользоваться списочным буфером для добавления элементов к его концу. Какому из них отдать предпочтение, зависит от конкретной ситуации. Обычно операция `::` хорошо поддается рекурсивным алгоритмам в стиле «разделяй и властвуй». Списочные буферы часто используются в более традиционном стиле, основанном на применении циклов.

Резюме

В этой главе была показана реализация списков в Scala. Список является наиболее востребованной структурой данных в Scala, с весьма совершенной реализацией. Оба имеющихся у класса `List` подкласса, `Nil` и `::`, относятся к case-классам. Но вместо рекурсивной обработки этой структуры многие основные методы работы со списками используют `ListBuffer`. В свою очередь класс `ListBuffer` реализован настолько аккуратно, что может эффективно создавать списки без выделения для своей работы дополнительного пространства памяти. Снаружи он обладает

свойствами функциональности, а внутри использует изменяемость для ускорения своей обычной работы, когда списочный буфер сбрасывается после вызова метода `toList`. Теперь, после изучения всех этих особенностей, классы списков вам знакомы как внутри, так и снаружи, а кроме этого, вы освоили пару хитрых приемов их реализации.

120 Точнее говоря, типы также позволяли бы методу head всегда вместо выдачи исключения входить в бесконечный цикл, но это явно не то, что нужно.

23. Возвращение к выражениям for

В главе 16 было показано, что функции высшего порядка, такие как `map`, `flatMap` и `filter`, предоставляют весьма эффективные конструкции для работы со списками. Но иногда уровень абстракции, требующийся этим функциям, усложняет понимание кода программ.

Рассмотрим пример. Предположим, что есть список людей, каждый элемент которого определен как экземпляр класса `Person`. У класса `Person` имеются поля, показывающие имя человека, его пол (мужской или женский) и его детей.

Определение класса имеет следующий вид:

```
scala> case class Person(name: String,
                         isMale: Boolean,
                         children: Person * )
```

А вот как выглядит список людей, взятых для примера:

```
val lara = Person("Lara", false)
val bob = Person("Bob", true)
val julie = Person("Julie", false, lara, bob)
val persons = List(lara, bob, julie)
```

Теперь предположим, что в списке нужно найти имена всех пар матерей и их детей. Используя `map`, `flatMap` и `filter`, можно сформулировать следующий запрос:

```
scala> persons filter (p => !p.isMale) flatMap (p
=>
                  (p.children map (c => (p.name,
c.name))))
res0: List[(String, String)] = List((Julie,Lara),
    (Julie,Bob))
```

Этот пример можно слегка оптимизировать, используя вместо `filter` вызов метода `withFilter`. Это позволит избавиться от создания промежуточной структуры данных для людей женского пола:

```
scala> persons withFilter (p => !p.isMale) flatMap
(p =>
                  (p.children map (c => (p.name,
c.name))))
res1: List[(String, String)] = List((Julie,Lara),
   (Julie,Bob))
```

Несмотря на то что эти запросы со своей задачей успешно справляются, для написания и понимания они не так уж просты. А есть ли более простой способ? Такой способ имеется. Помните выражения `for` из раздела 7.3? Используя выражение `for`, тот же самый пример можно записать следующим образом:

```
scala> for (p <- persons; if !p.isMale; c <-
p.children)
       yield (p.name, c.name)
res2: List[(String, String)] = List((Julie,Lara),
   (Julie,Bob))
```

Результат выполнения этого выражения окажется точно таким же, как и у предыдущего выражения. Более того, многим читателям кода выражение `for` будет понятнее предыдущего запроса, в котором использовались функции высшего порядка `map`, `flatMap` и `withFilter`.

Но может показаться, что последние два запроса отличаются друг от друга не так уж сильно. Фактически окажется, что компилятор Scala переведет второй запрос в первый. В более широком смысле компилятор переводит все выражения, выдающие результат с помощью инструкции `yield`, в сочетание вызовов функций высшего порядка `map`, `flatMap` и `withFilter`.

Все циклы `for` без инструкции `yield` переводятся в меньший по объему набор функций высшего порядка, в котором присутствуют только функции `withFilter` и `foreach`.

Сначала в этой главе будут рассмотрены определенные правила написания выражений `for`. После этого показано, как с их помощью можно упростить решение комбинаторных задач. И наконец, вы узнаете, как выполняется перевод выражений `for` и как в результате этого выражения `for` могут поспособствовать развитию языка Scala в новых областях применения.

23.1. Выражения for

В общем выражение `for` имеет следующий вид:

```
for ( seq ) yield expr
```

Здесь seq является последовательностью из *генераторов*, *определений* и *фильтров* с точками с запятой между стоящими друг за другом элементами. Выражение `for` показано в следующем примере:

```
for (p <- persons; n = p.name; if (n startsWith
"To"))
yield n
```

Это выражение `for` содержит один генератор, одно определение и один фильтр. Как упоминалось в разделе 7.3, последовательность можно заключить вместо круглых скобок в фигурные. Тогда точки с запятой можно будет не ставить:

```
for {
  p <- persons // генератор
  n = p.name   // определение
  if (n startsWith "To") // фильтр
```

```
} yield n
```

Генератор имеет следующую форму:

```
pat <- expr
```

Выражение expr обычно возвращает список, хотя чуть позже будет показано, что это обстоятельство можно обобщить. Шаблон pat поэлементно проверяется на соответствие всем элементам списка. При успешной проверке переменные в шаблоне привязываются к соответствующим частям элемента, как описано в главе 15. Но если проверка завершается неудачно, никакая ошибка `MatchError` не выдается. Вместо этого элемент просто выбрасывается из итерации.

В наиболее распространенном варианте использования шаблон pat представлен простой переменной x, как в выражении x <- expr. В данном случае переменная x просто используется при последовательном переборе всех элементов, возвращенных выражением expr.

Определение имеет форму

```
pat = expr
```

Это определение привязывает шаблон `pat` к переменной выражения expr. Таким образом, оно дает такой же эффект, что и определение `val`-переменной:

```
val x = expr
```

В наиболее распространенном варианте использования шаблон также представлен простой переменной x (например, x = expr). Тем самым *x* определяется в качестве имени для значения expr.

Фильтр имеет форму

```
if expr
```

Здесь expr является выражением, имеющим тип `Boolean`.

Фильтр выбрасывает из итерации все элементы, для которых выражение expr возвращает `false`.

Каждое выражение `for` начинается с генератора. Если в выражении `for` имеется сразу несколько генераторов, последующие генераторы изменяются быстрее предыдущих. Это легко проверить с помощью простого теста:

```
scala> for (x <- List(1, 2); y <- List("one",
"two"))
       yield (x, y)
res3: List[(Int, String)] =
  List((1,one), (1,two), (2,one), (2,two))
```

23.2. Задача N королев

Для выражений `for` наиболее подходящей областью применения являются комбинаторные головоломки. Примером такой головоломки является задача восьми королев: нужно на стандартной шахматной доске расставить восемь королев таким образом, чтобы ни одна из них не могла бить любую другую (королева может бить другую фигуру, если они находятся на одной вертикали, горизонтали или диагонали). Чтобы найти решение этой задачи, ее проще рассматривать в общем виде на шахматной доске произвольного размера. Тогда задача будет заключаться в размещении N королев на доске из $N \times N$ клеток, где N имеет произвольное значение. Давайте начнем нумеровать клетки с единицы, чтобы верхняя левая клетка доски размером $N \times N$ имела координаты (1, 1), а нижняя правая — (N, N).

Следует заметить, что для решения задачи N королев нужно поставить по королеве на каждую горизонталь. Можно последовательно расставлять королев по горизонталям, каждый раз проверяя, что только что поставленная на доску фигура не находится под ударом любых других ранее поставленных королев. В ходе проверки может оказаться, что королева, которую нужно

поставить на горизонталь k, будет под ударом на всех полях этой горизонтали от королев, установленных на горизонталях от 1 до k – 1. В таком случае следует прекратить данную часть проверки, чтобы продолжить с другой расстановкой королев на вертикалях от 1 до k – 1.

При императивном решении этой задачи королев будут ставить на доску по одной и они будут перемещаться по доске. Но, похоже, найти схему, которая реально перепробует все возможности, окажется нелегко. Более функциональный подход представляет собой решение непосредственно в виде значения. Решение состоит из списка координат, по одной для каждой поставленной на доску королевы. Но следует заметить, что найти за один шаг полноценное решение не удастся. Оно должно выстраиваться постепенно, по мере расстановки королев по последовательным строкам.

Это наводит на мысль о применении рекурсивного алгоритма. Предположим, что уже сгенерированы все решения по размещению k королев на доске размером $N \times N$ клеток, где $k < N$. Каждое такое решение может быть представлено в виде списка длиной k координат из строк и столбцов (`row`, `column`), где и `row`, и `column` являются числами в диапазоне от 1 до N. Эти списки частных решений удобнее будет рассматривать в виде стеков, где координаты королевы в строке k следуют первыми в списке, за ними идут координаты королевы в строке $k - 1$ и т. д. На дне стека находятся координаты королевы, помещенной на первую строку доски. Решение целиком представлено в виде списка списков, по одному элементу для каждого решения.

Теперь, чтобы поставить следующую королеву на строку $k + 1$, генерируются все возможные расширения каждого предыдущего решения на еще одну королеву. Это приводит к появлению еще одного списка, состоящего из списков решений, на этот раз длиной $k + 1$. Продолжайте процесс до получения всех решений для шахматной доски размером $N \times N$.

Этот алгоритмический замысел воплотился в функции placeQueens:

```
def queens(n: Int): List[List[(Int, Int)]] = {
  def placeQueens(k: Int): List[List[(Int, Int)]]
=
    if (k == 0)
      List(List())
    else
      for {
        queens <- placeQueens(k - 1)
        column <- 1 to n
        queen = (k, column)
        if isSafe(queen, queens)
       } yield queen :: queens
  placeQueens(n)
}
```

Внешняя функция queens в показанной ранее программе просто вызывает placeQueens с размером доски n, используемым в качестве аргумента. Задача функции-приложения placeQueens(k) состоит в генерировании в списке всех частных решений длиной k. Каждый элемент списка является одним решением, представленным списком длиной k. Следовательно, placeQueens возвращает список списков.

Если параметр k для placeQueens равен нулю, это означает, что нужно сгенерировать все решения, помещая нуль королев на нуль строк. Есть только одно такое решение: не ставить ни одной королевы. Оно представлено пустым списком. Следовательно, если параметр k равен нулю, placeQueens возвращает List(List()) — список, состоящий из одного элемента, являющегося пустым списком. Надо заметить, что это сильно отличается от пустого списка List(). Если placeQueens

возвращает `List()`, это означает отсутствие решений, а не единственное решение, состоящее из не установленной на доску королевы.

В иных случаях, когда `k` ≠ 0, вся работа `placeQueens` выполняется в выражении `for`. Первый генератор этого выражения `for` обходит все решения по установке на доске `k - 1` королев. Второй генератор обходит все возможные столбцы `column`, на которые можно поставить `k`-ю королеву. Третья часть выражения `for` определяет позицию новоиспеченной королевы, чтобы это была пара, состоящая из строки `k` и каждого выданного столбца `column`. Четвертая часть выражения `for` является фильтром, проверяющим с помощью метода `isSafe`, не находится ли новая королева под ударом предыдущих королев (определение `isSafe` будет рассмотрено чуть позже).

Если новая королева не находится под ударом любых других королев, может быть сформирована часть частного решения, стало быть, `placeQueens` с помощью выражения `queen :: queens` генерирует новое решение. Если новая королева не находится в безопасной от удара позиции, фильтр возвращает `false` и решение не генерируется.

Осталось только определить метод `isSafe`, используемый для проверки того, не находится ли рассматриваемая королева под ударом от любого другого элемента списка `queens`. Определение имеет следующий вид:

```
def isSafe(queen: (Int, Int), queens: List[(Int,
Int)]) =
  queens forall (q => !inCheck(queen, q))
def inCheck(q1: (Int, Int), q2: (Int, Int)) =
  q1._1 == q2._1 || // та же строка
  q1._2 == q2._2 || // тот же столбец
  (q1._1 - q2._1).abs == (q1._2 - q2._2).abs // на
диагонали
```

Метод `isSafe` выражает безопасность королевы относительно какой-либо из других королев. Метод `inCheck` показывает, что королевы `q1` и `q2` находятся под ударом друг друга.

Он возвращает `true` в одном из трех случаев.

Если две королевы имеют одну и ту же координату строки.

16. Если две королевы имеют одну и ту же координату столбца.

17. Если две королевы стоят на одной и той же диагонали (то есть разница между их строками и разница между их столбцами одинакова).

Первый случай, когда две королевы имеют одинаковую координату строки, в данном приложении невозможен, поскольку метод `placeQueens` уже позаботился о помещении каждой королевы на другую строку. Следовательно, эту проверку можно убрать, не изменяя функциональности программы.

23.3. Выполнение запросов с помощью выражений for

Система записи выражения `for`, по сути, эквивалентна самым распространенным операциям языков запросов к базам данных. Возьмем, к примеру, некую базу данных по имени `books`, представляющую собой список книг, где класс `Book` определен следующим образом:

```
case class Book(title: String, authors: String * )
```

Вот как выглядит небольшой фрагмент базы данных, представленный в виде списка в памяти:

```
val books: List[Book] =
  List(
```

```
    Book(
        "Structure and Interpretation of Computer
Programs",
      "Abelson, Harold", "Sussman, Gerald J."
    ),
    Book(
      "Principles of Compiler Design",
      "Aho, Alfred", "Ullman, Jeffrey"
    ),
    Book(
      "Programming in Modula-2",
      "Wirth, Niklaus"
    ),
    Book(
      "Elements of ML Programming",
      "Ullman, Jeffrey"
    ),
    Book(
       "The Java Language Specification", "Gosling,
James",
      "Joy, Bill", "Steele, Guy", "Bracha, Gilad"
    )
)
```

Чтобы найти заголовки всех книг, автор которых носит фамилию Gosling, нужен следующий код:

```
scala> for (b <- books; a <- b.authors
            if a startsWith "Gosling")
       yield b.title
res4: List[String] = List(The Java Language
Specification)
```

А чтобы найти заголовки всех книг, в которых встречается

строка Program, нужен такой код:

```
scala> for (b <- books if (b.title indexOf
"Program") >= 0)
       yield b.title
res5: List[String] = List(Structure and
Interpretation of
   Computer Programs, Programming in Modula-2,
Elements of ML
   Programming)
```

Чтобы в базе данных найти имена всех авторов, написавших хотя бы две книги, нужен следующий код:

```
scala> for (b1 <- books; b2 <- books if b1 != b2;
            a1 <- b1.authors; a2 <- b2.authors if
a1 == a2)
       yield a1
res6: List[String] = List(Ullman, Jeffrey, Ullman,
Jeffrey)
```

Последнее решение все же далеко от совершенства, поскольку в полученном в результате списке авторы будут фигурировать по нескольку раз. Нужно удалить повторяющихся авторов из списков. Добиться этого можно с помощью следующей функции:

```
scala>  def  removeDuplicates[A](xs:  List[A]):
List[A] = {
         if (xs.isEmpty) xs
         else
           xs.head :: removeDuplicates(
             xs.tail filter (x => x != xs.head)
           )
      }
removeDuplicates: [A](xs: List[A])List[A]
```

```
scala> removeDuplicates(res6)
res7: List[String] = List(Ullman, Jeffrey)
```

Стоит отметить, что последнее выражение в методе `removeDuplicates` может быть эквивалентно выражено путем использования `for`:

```
xs.head :: removeDuplicates(
  for (x <- xs.tail if x != xs.head) yield x
)
```

23.4. Трансляция выражений for

Каждое выражение `for` может быть представлено в понятиях трех функций высшего порядка: `map`, `flatMap` и `withFilter`. В этом разделе рассматривается схема трансляции, которая также используется компилятором Scala.

Трансляция выражений for с одним генератором

Предположим сначала, что у нас есть простое выражение `for`:

```
for ( x <- expr1 ) yield expr2
```

где x является переменной. Оно транслируется в следующее выражение:

```
expr1.map( x => expr2 )
```

Трансляция выражений for, начинающихся с генератора и фильтра

Теперь рассмотрим выражения `for`, сочетающие в себе ведущий генератор с некоторыми другими элементами. Выражение `for` следующей формы:

```
for ( x <- expr1 if expr2 ) yield expr3
```

транслируется в выражение:

```
for ( x <- expr1 withFilter ( x => expr2 )) yield
expr3
```

Эта трансляция выдает еще одно выражение `for`, которое короче исходного на один элемент, поскольку элемент `if` транслируется в применение `withFilter` к первому выражению генератора. Затем трансляция продолжается в отношении второго выражения, и в итоге получается следующий код:

```
expr1 withFilter ( x => expr2 ) map ( x => expr3 )
```

Точно такая же схема применима, если есть другие элементы, следующие за фильтром. Если seq является произвольной последовательностью генераторов, определений и фильтров, то выражение:

```
for ( x <- expr1 if expr2; seq) yield expr3
```

транслируется в выражение:

```
for ( x <- expr1 withFilter expr2; seq) yield
expr3
```

Затем процесс трансляции продолжается в отношении второго выражения, которое в очередной раз короче исходного на один элемент.

Трансляция выражений for, начинающихся с двух генераторов

Следующий вариант относится к обработке выражений `for`, начинающихся с двух генераторов, как в данном случае:

```
for ( x <- expr1; y <- expr2; seq) yield expr3
```

Здесь также предположим, что seq является произвольной последовательностью генераторов, определений и фильтров. Фактически элемент seq тоже может быть пустым, и тогда после $expr_2$ точки с запятой может не быть. Схема трансляции в любом случае будет оставаться прежней. Показанное ранее выражение `for` транслируется в выражение с применением метода `flatMap`:

```
expr1.flatMap( x => for ( y <- expr2; seq) yield
expr3 )
```

На этот раз получается еще одно выражение `for`, которое находится в переданном методу `flatMap` значении функции. Это выражение `for`, которое опять проще исходного на один элемент, транслируется с применением тех же самых правил.

Трех показанных схем достаточно для трансляции всех выражений `for`, содержащих только генераторы и фильтры, где генераторы привязывают лишь простые переменные. Возьмем, к примеру, запрос из раздела 23.3 «найти имена всех авторов, написавших хотя бы две книги»:

```
for (b1 <- books; b2 <- books if b1 != b2;
      a1 <- b1.authors; a2 <- b2.authors if a1 ==
a2)
yield a1
```

Этот запрос транслируется в следующую комбинацию `map`/`flatMap`/`withFilter`:

```
books flatMap (b1 =>
  books withFilter (b2 => b1 != b2) flatMap (b2 =>
    b1.authors flatMap (a1 =>
       b2.authors withFilter (a2 => a1 == a2) map
(a2 =>
```

```
        a1))))
```

Представленная до сих пор схема трансляции пока не справляется с генераторами, привязывающими не простые переменные, а целые шаблоны. Она также пока не охватывает определения. Эти два аспекта будут рассмотрены в следующих двух подразделах.

Трансляция шаблонов в генераторах

Если в левой части генератора находится не простая переменная, а шаблон, *pat*, схема трансляции усложняется. Тот случай, когда в выражении `for` имеется привязка кортежа переменных, обрабатывается довольно просто. При этом применяется почти такая же схема, как и при одной переменной.

Выражение `for`, имеющее следующую форму:

```
for (( x1 , ..., xn ) <- expr1 ) yield expr2
```

транслируется в выражение:

```
expr1.map { case ( x1 , ..., xn ) => expr2 }
```

Ситуация усложняется, если в левой части генератора находится не одна переменная и не кортеж переменных, а произвольный шаблон pat. В таком случае выражение:

```
for (pat <- expr1 ) yield expr2
```

транслируется в выражение:

```
expr1 withFilter {
  case pat => true
  case _ => false
} map {
```

```
  case pat => expr2
}
```

То есть сначала генерируемые элементы фильтруются, и только потом выполняется отображение этого шаблона соответствия pat. Тем самым гарантируется, что генератор, основанный на соответствии шаблону, никогда не выдаст ошибку `MatchError`.

В представленной здесь схеме рассматриваются только случаи, когда выражение `for` содержит лишь один генератор, основанный на соответствии шаблону. Аналогичные правила применяются и тогда, когда в выражении `for` содержатся другие генераторы, фильтры или определения. Поскольку эти дополнительные правила не помогают лучше понять тему, здесь они не рассматриваются. Любопытные читатели могут найти их в публикации Scala Language Specification121.

Трансляция определений

Еще одна пропущенная ситуация касается содержания в выражении `for` встроенных определений. Рассмотрим типичный вариант:

```
for (x <- expr1 ; y = expr2; seq) yield expr3
```

Предположим в очередной раз, что элемент seq (возможно, пустой) является последовательностью генераторов, определений и фильтров. Это выражение транслируется в следующее выражение:

```
for ((x, y) <- for (x <- expr1 ) yield (x, expr2);
seq)
yield expr3
```

Как видите, $expr_2$ вычисляется при каждой генерации нового

значения x. Необходимость многократного вычисления обусловливается тем, что $expr_2$ может ссылаться на *x* и поэтому для изменения значений x должно быть вычислено заново. Вы как программист можете прийти к заключению, что встраивать определения в выражения `for`, которые не ссылаются на переменные, привязанные к какому-нибудь предшествующему генератору, вряд ли разумно, поскольку многократные вычисления таких выражений повлекут за собой существенные издержки. Например, вместо:

```
for (x <- 1 to 1000; y = expensiveComputationNotInvolvingX)
yield x * y
```

во многих случаях лучше воспользоваться следующим кодом:

```
val y = expensiveComputationNotInvolvingX
for (x <- 1 to 1000) yield x * y
```

Трансляция, применяемая для циклов

В предыдущих подразделах было показано, как транслируется выражение `for`, содержащее оператор возвращения `yield`. А как насчет циклов, которые просто дают побочный эффект, не возвращая вообще ничего? Их трансляция выполняется аналогично, но она проще, чем для выражений `for`. В принципе, там, где предыдущая схема превращения использовала `map` или `flatMap`, в схеме превращения для циклов `for` используется просто `foreach`.

Например, выражение:

```
for (x <- expr1) body
```

транслируется в:

```
expr1 foreach ( x => body).
```

Более объемным примером будет выражение:

```
for (x <- expr1; if expr2; y <- expr3) body
```

Оно транслируется в следующий код:

```
expr1 withFilter (x => expr2) foreach (x =>
  expr3 foreach (y => body))
```

Например, приведенное далее выражение суммирует все элементы матрицы, представленной в виде списка списков:

```
var sum = 0
for (xs <- xss; x <- xs) sum += x
```

Этот цикл транслируется в два вложенных применения `foreach`:

```
var sum = 0
xss foreach (xs =>
  xs foreach (x =>
    sum += x))
```

23.5. Движение в обратном направлении

В предыдущем разделе было показано, что выражения `for` могут транслироваться в использование функций высшего порядка `map`, `flatMap` и `withFilter`. Но можно пойти и в обратном направлении и каждое применение `map`, `flatMap` или `withFilter` представить в виде выражения `for`.

Рассмотрим реализацию этих трех методов в понятиях выражений `for`. Методы содержатся в объекте `Demo`, чтобы их можно было отличить от стандартных операций над объектами

типа List. Точнее, все три функции получают в качестве параметра значение типа List, но схема трансляции будет успешно работать и с другими типами коллекций:

```
object Demo {
  def map[A, B](xs: List[A], f: A => B): List[B] =
    for (x <- xs) yield f(x)
  def flatMap[A, B](xs: List[A], f: A => List[B]):
List[B] =
    for (x <- xs; y <- f(x)) yield y
   def filter[A](xs: List[A], p: A => Boolean):
List[A] =
    for (x <- xs if p(x)) yield x }
```

Думаю, у вас не вызовет удивления то, что трансляция выражения for, использованного в теле метода Demo.map, превратится в вызов map в классе List. По аналогии с этим Demo.flatMap и Demo.filter будут переведены в flatMap и withFilter в классе List. Таким образом, эта небольшая демонстрация показывает, что выражения for действительно эквивалентны по своей выразительности применению функций map, flatMap и withFilter.

23.6. Обобщение сведений о выражениях for

Поскольку трансляция выражений for зависит только от наличия методов map, flatMap и withFilter, систему записи выражений for можно применить к большому классу типов данных.

Вам уже встречалось применение выражений for в отношении списков и массивов. Их поддержка обусловливается тем, что в списках, как и в массивах, определены операции map, flatMap и withFilter. Поскольку в них также определен метод foreach, при работе с этими типами данных доступны и циклы for.

Кроме списков и массивов, в стандартной библиотеке Scala есть множество других типов, поддерживающих те же четыре метода, а следовательно, позволяющих использовать выражения `for`. В качестве примеров можно назвать диапазоны, итераторы, потоки и все реализации наборов. Также вполне возможно путем определения всех необходимых методов получить поддержку выражений `for` для ваших собственных типов данных. Для поддержки всего диапазона выражений `for` и циклов `for` в качестве методов вашего типа данных понадобятся определения методов `map`, `flatMap`, `withFilter` и `foreach`. Но можно также определить поднабор этих методов и, соответственно, получить поднабор всех возможных выражений и циклов `for`.

При этом нужно придерживаться следующих строгих правил.

- Если в вашем типе определяется только метод `map`, можно будет применять выражения `for`, содержащие только один генератор.
- Если наряду с `map` определяется `flatMap`, можно будет применять выражения `for`, содержащие несколько генераторов.
- Если определяется метод `foreach`, можно будет применять циклы `for` (как с одним, так и с несколькими генераторами).
- Если определяется метод `withFilter`, можно будет применять выражения `for` с фильтром, где выражение `for` начинается с `if`.

Трансляция выражений `for` происходит до проверки соответствия типов. Тем самым допускается максимальная гибкость, поскольку требуется, чтобы проверку типов прошел лишь результат расширения выражения `for`. Для самих выражений `for` правила типизации в Scala не определяются, кроме того, не требуется наличия какой-либо конкретной сигнатуры типов в методах `map`, `flatMap`, `withFilter` или `foreach`.

Тем не менее существует стандартная настройка, отражающая

наиболее распространенный замысел, вкладываемый в методы высшего порядка, в которые транслируются выражения `for`. Предположим, что имеется некий параметризованный класс `C`, который, как правило, будет представлять некую разновидность коллекции. Тогда вполне естественно будет выбрать для `map`, `flatMap`, `withFilter` и `foreach` следующие сигнатуры типа:

```
abstract class C[A] {
  def map[B](f: A => B): C[B]
  def flatMap[B](f: A => C[B]): C[B]
  def withFilter(p: A => Boolean): C[A]
  def foreach(b: A => Unit): Unit
}
```

То есть функция `map` получает функцию от типа элементов коллекции `A` к некоторому другому типу `B`. Она создает новую коллекцию, однотипную с `C`, но с `B` в качестве типа элементов. Метод `flatMap` получает функцию `f` от `A` к некой `C`-коллекции из `B`-элементов и создает `C`-коллекцию из `B`-элементов. Метод `withFilter` получает функцию-предикат от типа элементов коллекции `A` к `Boolean`. Он создает коллекцию того же типа, что и та, в отношении которой он был вызван. И наконец, метод `foreach` получает функцию от `A` к `Unit` и создает результат типа `Unit`.

В показанном ранее классе `C` метод `withFilter` создает новую коллекцию того же класса. Этот означает, что при каждом вызове `withFilter` появляется новый объект типа `C`, совпадающего с тем типом, с которым будет работать фильтр. Теперь в трансляции выражений `for` за любым вызовом `withFilter` всегда будет следовать вызов одного из трех других методов. Поэтому объект, вводимый методом `withFilter`, будет сразу же разобран одним из других методов. Если объекты класса `C` велики (например, это длинные последовательности), может потребоваться обойти

создание такого промежуточного объекта. В качестве стандартного приема можно позволить методу `withFilter` возвращать не объект `C`, а просто объект-оболочку, который помнит, что элементы перед последующей обработкой должны быть отфильтрованы.

Концентрируясь только на первых трех функциях класса `C`, нужно отметить следующие факты. В функциональном программировании существует общее понятие *монада*, с помощью которого можно объяснить большое количество типов с вычислениями, начиная от коллекций и заканчивая, если называть только некоторые из них, вычислениями с состоянием и вводом-выводом, обратными вычислениями и транзакциями. Функции `map`, `flatMap` и `withFilter` можно сформулировать в понятиях монад, и если сделать это, то все сведется к тому, что у них будут точно такие же типы, что и здесь.

Более того, каждую монаду можно охарактеризовать с помощью `map`, `flatMap` и `withFilter`, если прибавить к этому блочный конструктор, создающий монаду из значения элемента. В объектно-ориентированных языках этот блочный конструктор является просто конструктором экземпляра или фабричным методом. Поэтому методы `map`, `flatMap` и `withFilter` могут выступать как объектно-ориентированные версии функциональной концепции монад. Поскольку выражения `for` эквивалентны применению этих трех методов, они могут рассматриваться в качестве синтаксиса для монад.

Все это говорит о том, что концепция выражения `for` более универсальна, чем просто обход элементов коллекции, и это действительно так. Например, выражения `for` также играют важную роль в асинхронном вводе-выводе или могут использоваться в качестве альтернативной системы записи для значений, которые по каким-то причинам отсутствуют. В библиотеках Scala проследите за появлениями `map`, `flatMap` и `withFilter`: там, где они присутствуют, сами собой

напрашиваются выражения `for`, которые могут стать весьма лаконичным способом работы с элементами типа.

Резюме

В этой главе было предложено заглянуть во внутреннюю кухню выражений и циклов `for`. Вы узнали, что они транслируются в применения стандартного набора методов высшего порядка. В результате было показано, что выражения `for` фактически имеют намного более универсальное применение, чем простой обход элементов коллекций, и что у вас есть возможность создавать для их поддержки собственные классы.

121 Odersky M. The Scala Language Specification, Version 2.9. EPFL, May 2011. Available on the web at http://www.scalalang.org/docu/manuals.html (accessed April 20, 2014).

24. Углубленное изучение коллекций

В Scala включена весьма интересная и мощная библиотека коллекций. Хотя на первый взгляд API коллекций представляется незаметным, вызванные им изменения в вашем стиле программирования могут быть весьма существенными. Зачастую это похоже на работу на высоком уровне с основными строительными блоками программы, представляющими собой скорее коллекции, чем их элементы. Этот новый стиль программирования требует некоторой адаптации. К счастью, ее облегчает ряд привлекательных свойств коллекций Scala. Они просты в использовании, лаконичны в описании, безопасны, быстры в работе и универсальны.

- **Простота использования.** Небольшого словаря, содержащего от 20 до 50 методов, вполне достаточно для решения основного набора задач всего за пару операций. Не нужно морочить голову сложными цикличными структурами или рекурсиями. Стабильные коллекции и операции без побочных эффектов означают, что вам не следует опасаться случайного повреждения существующих коллекций новыми данными. Взаимовлияние итераторов и обновление коллекций исключены.

- **Лаконичность.** То, что раньше занимало от одного до нескольких циклов, теперь можно выразить всего одним словом. Можно выполнять функциональные операции с применением упрощенного синтаксиса и без особых усилий комбинировать операции таким образом, чтобы в результате получалось нечто похожее на обычные алгебраические формулы.

- **Безопасность.** Чтобы разобраться в этом вопросе, нужен определенный опыт. Природа коллекций Scala с их статической типизацией и функциональностью означает, что подавляющее

большинство потенциальных ошибок отлавливается еще на стадии компиляции. Это достигается благодаря следующему:

- сами операции над коллекциями используются довольно широко, а следовательно, прошли проверку временем;
- использование операций над коллекциями обусловливает ввод и вывод в виде параметров функций и результатов;
- такие явные входные и выходные данные являются предметом для проверки соответствия статических типов.

Суть заключается в том, что большинство случаев неверного использования кода проявится в виде ошибок соответствия типа. И вовсе не редкость, когда программы, состоящие из нескольких сотен строк кода, запускаются с первой же попытки.

- **Скорость.** Операции с коллекциями, имеющиеся в библиотеках, уже настроены и оптимизированы. В результате этого использование коллекций обычно отличается высокой эффективностью. Тщательно настроенные структуры данных и операции могут улучшить ситуацию, но в то же время, принимая решения, далекие от оптимальных, можно ее значительно ухудшить. Более того, коллекции были адаптированы под параллельную обработку на многоядерных системах. Параллельно обрабатываемые коллекции поддерживают те же самые операции, что и последовательно обрабатываемые, поэтому изучать новые операции и переписывать код не нужно. Последовательно обрабатываемые коллекции можно превратить в параллельно обрабатываемые просто вызовом метода `par`.

- **Универсальность.** Коллекции реализуют одни и те же операции над любым типом там, где в этих операциях есть определенный смысл. Следовательно, используя сравнительно небольшой словарь операций, вы приобретаете множество возможностей.

Например, концептуально строка является последовательностью символов. Следовательно, в коллекциях Scala строки поддерживают все операции с последовательностями. То же самое справедливо и для массивов.

В этой главе дается углубленное описание API имеющихся в Scala классов коллекций с точки зрения их использования. Краткий тур по библиотекам коллекций был представлен в главе 17. В этой главе предстоит поучаствовать в более подробном путешествии, в ходе которого будут показаны все классы коллекций и все определенные в них методы, то есть в него будут включены все сведения, необходимые для использования коллекций Scala. Забегая вперед: для тех, кто собирается заниматься реализацией новых типов коллекций, в главе 25 основное внимание будет уделено вопросам архитектуры библиотек и возможностям их расширения.

24.1. Изменяемые и неизменяемые коллекции

Как вам уже известно, в целом коллекции в Scala подразделяются на изменяемые и неизменяемые. Изменяемые коллекции могут обновляться или расширяться на месте. Это означает возникновение в качестве побочного эффекта возможности изменения, добавления или удаления элементов коллекции. В отличие от этого, неизменяемые коллекции всегда сохраняют неизменный вид. Но все же есть операции, имитирующие добавление, удаление или обновление, и каждая из таких операций приводит к возвращению новой коллекции, оставляя старую без изменений.

Все классы коллекций находятся в пакете `scala.collection` или в одном из его подпакетов: `mutable`, `immutable` и `generic`. Большинство классов коллекций, востребованных в клиентском коде, существуют в трех вариантах, которые имеют разные

характеристики в смысле возможности изменения. Эти три варианта находятся в пакетах `scala.collection`, `scala.collection.immutable` и `scala.collection.mutable`.

Коллекция в пакете `scala.collection.immutable` гарантированно неизменяемая ни с одной из сторон. Такая коллекция после создания всегда остается неизменной. Поэтому можно полагаться на то, что многократные обращения к одному и тому же значению коллекции в разные моменты времени всегда будут выдавать коллекцию с одинаковыми элементами.

Коллекция в пакете `scala.collection.mutable` известна тем, что у нее имеется ряд операций, изменяющих коллекцию на месте. Эти операции позволяют создавать код, изменяющий саму коллекцию. Но при этом нужно четко понимать, что существует вероятность обновления из других частей исходного кода, и защищаться от нее.

Коллекции в пакете `scala.collection` могут быть как изменяемыми, так и неизменяемыми. Например, `scala.collection.IndexedSeq[T]` является родительским трейтом как для `scala.collection.immutable.IndexedSeq[T]`, так и для его изменяемого родственника `scala.collection.mutable.IndexedSeq[T]`. В целом корневые коллекции в пакете `scala.collection` определяют такой же интерфейс, как и у неизменяемых коллекций. И обычно к изменяемым коллекциям в пакете `scala.collection.mutable` добавляются некоторые операции модификаций, производимых за счет сторонних эффектов.

Разница между корневыми и неизменяемыми коллекциями состоит в следующем: клиенты неизменяемых коллекций получают гарантии того, что никто не сможет внести в коллекцию изменения, а клиенты корневых коллекций знают только, что не могут изменить коллекцию самостоятельно. Даже если статический тип такой коллекции не предоставляет операций для

ее изменения, все же существует вероятность того, что в процессе выполнения программы она получит тип изменяемой коллекции, предоставив другим клиентам возможность внесения в нее изменений.

По умолчанию в Scala всегда выбираются неизменяемые коллекции. Например, если просто написать `Set` без какого-либо префикса или ничего не импортируя, будет получен неизменяемый набор, а если написать `Iterable`, будет получен неизменяемый объект с возможностью обхода его элементов, поскольку имеются исходные привязки, импортируемые из пакета `scala`. Для получения исходных изменяемых версий следует явно указать `collection.mutable.Set` или `collection.mutable.Iterable`.

Последним пакетом в иерархии коллекций является `collection.generic`. В этом пакете содержатся строительные блоки для реализации коллекций. Обычно коллекции классов полагают, что реализация некоторых их операций находится в классах пакета `generic`. Впрочем, постоянные пользователи коллекций должны ссылаться на классы в пакете `generic` только в крайних случаях.

24.2. Согласованность коллекций

Наиболее важные классы коллекций показаны на рис. 24.1. Между этими классами есть много общего. Например, каждая разновидность коллекции может быть создана с использованием единого унифицированного синтаксиса, заключающегося в записи имени класса коллекции, за которым стоят элементы этой коллекции:

```
Traversable(1, 2, 3)
Iterable("x", "y", "z")
Map("x" -> 24, "y" -> 25, "z" -> 26)
```

```
Set(Color.Red, Color.Green, Color.Blue)
SortedSet("hello", "world")
Buffer(x, y, z)
IndexedSeq(1.0, 2.0)
LinearSeq(a, b, c)
```

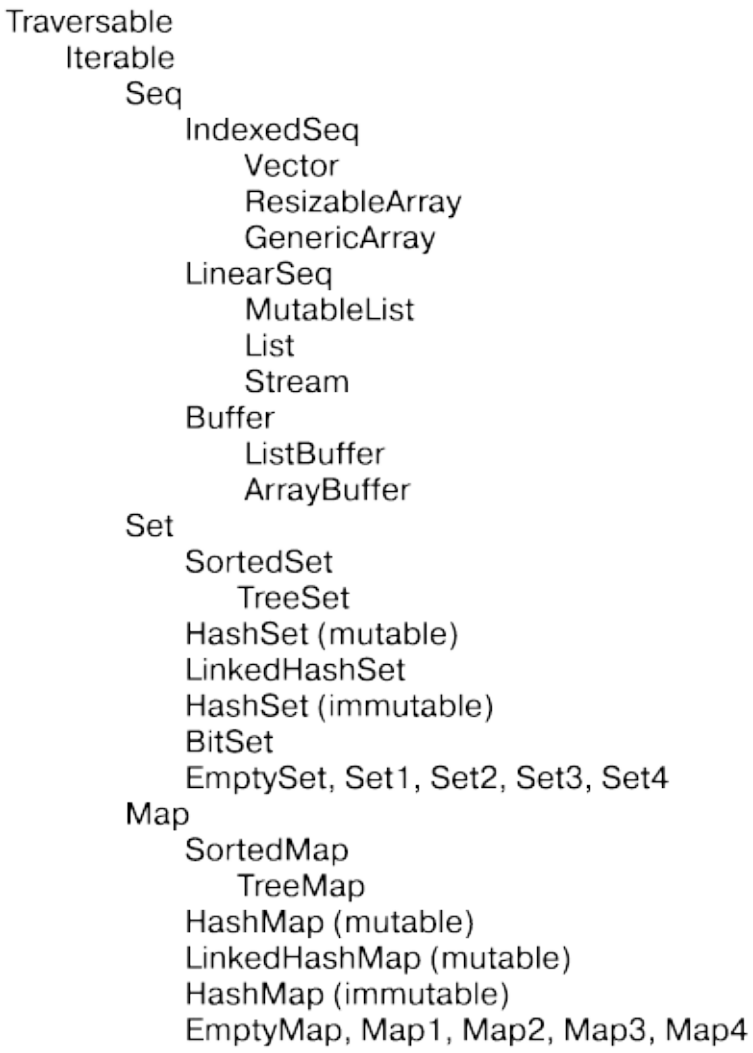


Рис. 24.1. Иерархия коллекций

Тот же принцип применяется и к конкретным реализациям коллекций:

```
List(1, 2, 3)
HashMap("x" -> 24, "y" -> 25, "z" -> 26)
```

Методы `toString` для всех коллекций выводят такую же информацию, которая показана ранее, с именем типа, за которым следуют значения элементов коллекции, заключенные в круглые скобки. Все коллекции поддерживают API, предоставляемый `Traversable`, но все их методы возвращают свой собственный класс, а не корневой класс `Traversable`. Например, у метода `map` класса `List` тип возвращаемого значения `List`, у метода `map` класса `Set` тип возвращаемого значения `Set`. То есть статический возвращаемый тип этих методов абсолютно точен:

```
scala> List(1, 2, 3) map (_ + 1)
res0: List[Int] = List(2, 3, 4)

scala> Set(1, 2, 3) map (_ * 2)
res1: scala.collection.immutable.Set[Int] = Set(2,
4, 6)
```

Эквивалентность для всех классов коллекций также организована по единому принципу. Более подробно этот вопрос рассматривается в разделе 24.13.

Большинство классов, показанных на рис. 24.1, существуют в трех вариантах: корневом, изменяемом и неизменяемом (root, mutable и immutable). Единственным исключением является трейт `Buffer`, существующий только в виде изменяемой коллекции.

Далее в главе все эти классы будут рассмотрены поочередно.

24.3. Трейт Traversable

На вершине иерархии коллекций находится трейт `Traversable`. Его единственной абстрактной операцией является `foreach`:

```
def foreach[U](f: Elem => U)
```

Классам коллекций, реализующим `Traversable`, нужно

определить только этот метод, а все остальные методы могут быть унаследованы из `Traversable`.

Метод `foreach` предназначен для обхода всех элементов коллекции и применения к каждому из них определенной операции `f`. Тип этой операции обозначен как `Elem => U`, где `Elem` является типом элементов коллекции, а `U` — произвольным типом результата. Вызов `f` выполняется только с целью получения побочного эффекта — фактически `foreach` отбрасывает любой результат, выдаваемый `f`.

В `Traversable` также определяется много конкретных методов (перечислены в табл. 24.1). Их можно разбить на следующие категории.

- **Сложение** `++`, складывает два найденных при обходе элемента или прибавляет все элементы итератора к уже пройденным элементам.

- **Операции отображения** `map`, `flatMap` и `collect`, которые создают новую коллекцию путем применения функции к элементам коллекции.

- **Преобразования** `toIndexedSeq`, `toIterable`, `toStream`, `toArray`, `toList`, `toSeq`, `toSet` и `toMap`, которые превращают коллекцию с возможностью обхода элементов в более конкретную коллекцию. Все эти преобразования возвращают - объект-получатель, если он уже соответствует требуемому типу коллекции. Например, применение `toList` к списку выдаст сам список.

- **Операции копирования** `copyToBuffer` и `copyToArray`. Их названия говорят, что они копируют элементы коллекции в буфер или массив соответственно.

- **Операции определения размера** `isEmpty`, `nonEmpty`, `size` и

hasDefiniteSize. Коллекции с возможностью обхода их элементов могут быть конечными или бесконечными. Примером бесконечной обходимой коллекции является поток натуральных чисел Stream.from(0). Метод hasDefiniteSize показывает потенциально возможную бесконечность коллекции. Если метод hasDefiniteSize возвращает true, значит коллекция, безусловно, конечная. Если этот метод возвращает false, то коллекция может быть бесконечной и тогда метод size выдаст ошибку или не возвратит никакого значения.

- **Операции извлечения элементов** head, last, headOption, lastOption и find. Выбирают первый или последний элемент коллекции или же первый элемент, соответствующий условию. Следует, однако, заметить, что не у всех коллекций имеется четко определенное значение первого и последнего. Например, хеш-набор может хранить элементы в соответствии с их хеш-ключами, и порядок их следования от запуска к запуску может изменяться. В таком случае для разных запусков программы первый элемент хеш-набора также может быть разным. Коллекция считается упорядоченной, если она всегда выдает свои элементы в одном и том же порядке. Большинство коллекций упорядоченные, но некоторые, такие как хеш-наборы, таковыми не являются, отказ от упорядочения добавляет им эффективности. Упорядочение зачастую необходимо, чтобы получать воспроизводимые тесты и способствовать отладке. Поэтому коллекции Scala предоставляют упорядоченные альтернативы для всех типов коллекций. Например, упорядоченной альтернативой для HashSet является LinkedHashSet.

- **Операции извлечения подколлекций** takeWhile, tail, init, slice, take, drop, filter, dropWhile, filterNot и withFilter. Все эти операции возвращают подколлекцию,

определяемую диапазоном индексов или предикатом.

- **Операции подразделения** `splitAt`, `span`, `partition` и `groupBy`, которые разбивают элементы переданной коллекции на несколько подколлекций.

- **Операции тестирования элементов** `exists`, `forall` и `count`, которые тестируют элементы коллекции с заданным предикатом.

- **Операции свертки** `foldLeft`, `foldRight`, `/:`, `:\`, `reduceLeft`, `reduceRight`, которые применяют бинарные операции к следующим друг за другом элементам.

- **Операции специализированных сверток** `sum`, `product`, `min` и `max`, которые работают с коллекциями конкретных типов (числовыми или аналогичными им).

- **Операции со строками** `mkString`, `addString` и `stringPrefix`, которые предоставляют альтернативные способы преобразования коллекции в строку.

- **Операции представлений**, которые состоят из двух перегружаемых вариантов метода `view`. Операция `view` вычисляется в «ленивом» режиме. Более подробно операции просмотра будут рассмотрены в разделе 24.14.

Таблица 24.1. Операции в трейте Traversable

Что собой представляет	Чем занимается
Абстрактный метод xs foreach f	Применяет функцию f к каждому элементу xs
Сложение xs ++ ys	Создает коллекцию, состоящую из элементов как xs, так и ys. При этом ys должен быть коллекцией TraversableOnce, то есть либо Traversable, либо Iterator

Отображение: xs map f	Создает коллекцию, получающуюся в результате применения функции f к каждому элементу в xs.
xs flatMap f	Создает коллекцию, получающуюся в результате применения относящейся ко всей коллекции функции f к каждому элементу в xs, и объединяет результаты.
xs collect f	Создает коллекцию, получающуюся в результате использования частично применяемой функции f для каждого элемента в xs, для которого она определена, и объединяет результаты
Преобразование: xs.toArray	Превращает коллекцию в массив.
xs.toList	Превращает коллекцию в список.
xs.toIterable	Превращает коллекцию в итерируемую коллекцию.
xs.toSeq	Превращает коллекцию в последовательность.
xs.toIndexedSeq	Превращает коллекцию в проиндексированную последовательность.
xs.toStream	Превращает коллекцию в поток («лениво» вычисляемую последовательность).
xs.toSet	Превращает коллекцию в набор.
xs.toMap	Превращает коллекцию пар «ключ — значение» в отображение
Копирование: xs copyToBuffer buf	Копирует все элементы коллекции в буфер buf.
xs copyToArray(arr, s, len)	Копирует максимум len элементов, имеющихся в массиве arr, начиная с индекса s. Последние два аргумента являются необязательными
Получение информации о размере: xs.isEmpty	Проверяет коллекцию на пустоту.
xs.nonEmpty	Проверяет коллекцию на содержание элементов.
xs.size	Возвращает количество элементов в коллекции.
xs.hasDefiniteSize	Возвращает true, если известно, что xs конечного размера
Извлечение элементов: xs.head	Возвращает первый элемент коллекции (или какой-нибудь элемент, если порядок следования элементов не определен).
xs.headOption	Возвращает первый элемент xs в Option-значении или None, если xs пуст
Продолжение^	

Таблица 24.1 (продолжение)

Что собой представляет	Чем занимается
xs.lastOption	Возвращает последний элемент xs в Option-значении параметра или None, если xs пуст.
xs find p	Возвращает Option-значение, содержащее первый элемент в xs, удовлетворяющий условию p, или None, если ни один элемент не определен
Создание подколлекций: xs.tail	Возвращает всю коллекцию, за исключением xs.head.
xs.init	Возвращает всю коллекцию, за исключением xs.last.
xs slice (from, to)	Возвращает коллекцию, состоящую из элементов в некотором диапазоне индексов xs (от from до to, исключая последний).
xs take n	Возвращает коллекцию, состоящую из первых n элементов xs (или какие-либо произвольные элементы в количестве n, если порядок следования элементов не определен).
xs drop n	Возвращает всю коллекцию xs, за вычетом тех элементов, которые получаются при использовании выражения xs take n.
xs takeWhile p	Возвращает самый длинный префикс элементов в коллекции, каждый из которых соответствует условию p.
xs dropWhile p	Возвращает коллекцию самого длинного префикса элементов, каждый из которых соответствует условию p.
xs filter p	Возвращает коллекцию, состоящую только из тех элементов xs, которые соответствуют условию p.
xs withFilter p	Нестрогий фильтр коллекции. Все операции в отношении фильтруемых элементов будут применяться только к тем элементам xs, для которых условие p вычисляется в true.
xs filterNot p	Возвращает коллекцию, состоящую из тех элементов xs, которые не соответствуют условию p
Разбиение на подразделы: xs splitAt n	Разбивает xs по позиции на пару коллекций (xs take n, xs drop n).
xs span p	Разбивает xs в соответствии с предикатом на пару коллекций (xs takeWhile p, xs.dropWhile p).
xs partition p	Разбивает xs на пару коллекций, в одной из которых находятся все элементы, удовлетворяющие условию p, а в другой — все элементы, не удовлетворяющие этому условию (xs filter p, xs.filterNot p).
xs groupBy f	Разбивает xs на отображение коллекций в соответствии с дискриминаторной функцией f

Состояния элементов: xs forall p	Выдает булево значение, показывающее, соблюдается ли условие p для всех элементов xs.
xs exists p	Выдает булево значение, показывающее, соблюдается ли условие p для каких-либо элементов xs.
Что собой представляет	**Чем занимается**
xs count p	Возвращает количество элементов в xs, удовлетворяющих условию p
Свертки: (z /: xs)(op)	Применяет бинарную операцию op между последовательными элементами xs, продвигаясь слева направо, начиная с z.
(xs :\ z)(op)	Применяет бинарную операцию op между последовательными элементами xs, продвигаясь справа налево, начиная с z.
xs.foldLeft(z)(op)	То же самое, что и (z /: xs)(op).
xs.foldRight(z)(op)	То же самое, что и (xs :\ z)(op).
xs reduceLeft op	Применяет бинарную операцию op между последовательными элементами xs непустой коллекции xs, продвигаясь слева направо.
xs reduceRight op	Применяет бинарную операцию op между последовательными элементами xs непустой коллекции xs, продвигаясь справа налево
Специализированные свертки: xs.sum	Возвращает сумму числовых элементов коллекции xs.
xs.product	Возвращает произведение числовых элементов коллекции xs.
xs.min	Возвращает минимальное значение упорядоченных элементов коллекции xs.
xs.max	Возвращает максимальное значение упорядоченных элементов коллекции xs
Строки: xs addString (b, start, sep, end)	Добавляет строку к строковому буферу b типа StringBuilder со всеми элементами xs между разделителями sep, заключенными в строки start и end. Аргументы start, sep и end являются необязательными.
xs mkString (start, sep, end)	Превращает коллекцию в строку со всеми элементами xs между разделителями sep, заключенными в строки start и end. Аргументы start, sep и end являются необязательными.
xs.stringPrefix	Вставляет имя коллекции перед строкой, возвращенной операцией xs.toString
Представления: xs.view	Создает представление xs.
	Создает представление, показывающее элементы xs в указанном

xs view (from, to)	диапазоне индексов

24.4. Трейт Iterable

Следующим трейтом, находящимся на вершине схемы, показанной на рис. 24.1, является `Iterable`. Все методы в этом трейте определены в понятиях абстрактного метода `iterator`, который поочередно выдает элементы коллекции. Абстрактный метод `foreach`, унаследованный из трейта `Traversable`, реализуется в `Iterable` в понятиях метода `iterator`. Фактическая реализация выглядит следующим образом:

```
def foreach[U](f: Elem => U): Unit = {
  val it = iterator
  while (it.hasNext) f(it.next())
}
```

Эта стандартная реализация `foreach` в `Iterable` переопределяется довольно многими подклассами класса `Iterable`, поскольку ими может быть обеспечена более эффективная реализация. Следует помнить, что `foreach` является основой реализации всех операций в `Traversable`, поэтому производительность этого метода очень важна.

В `Iterable` имеются еще два метода, возвращающие итераторы: `grouped` и `sliding`. Но эти итераторы возвращают не отдельные элементы, а целые подпоследовательности элементов исходной коллекции. Максимальный размер подпоследовательностей задается в виде аргумента этих методов. Метод `grouped` делит элементы коллекции на блоки по нарастающей, а метод `sliding` выдает элементы блоками, попадающими в скользящее по коллекции окно. Разобраться, чем они отличаются друг от друга, можно, рассмотрев следующий сеанс работы с интерпретатором:

```
scala> val xs = List(1, 2, 3, 4, 5)
xs: List[Int] = List(1, 2, 3, 4, 5)

scala> val git = xs grouped 3
git: Iterator[List[Int]] = non-empty iterator

scala> git.next()
res2: List[Int] = List(1, 2, 3)

scala> git.next()
res3: List[Int] = List(4, 5)

scala> val sit = xs sliding 3
sit: Iterator[List[Int]] = non-empty iterator

scala> sit.next()
res4: List[Int] = List(1, 2, 3)

scala> sit.next()
res5: List[Int] = List(2, 3, 4)

scala> sit.next()
res6: List[Int] = List(3, 4, 5)
```

Трейт `Iterable` также добавляет к `Traversable` некоторые другие методы, которые могут быть эффективно реализованы только при доступности итератора. Все эти методы сведены в табл. 24.2.

Таблица 24.2. Операции в трейте Iterable

Что собой представляет	Чем занимается

Абстрактный метод xs.iterator	Создает итератор, выдающий каждый элемент, имеющийся в xs, в том же порядке, в котором обходит элементы foreach
Другие итераторы: xs grouped size	Создает итератор, выдающий из коллекции блоки фиксированного размера.
xs sliding size	Создает итератор, выдающий из коллекции блоки, попадающие в скользящее по коллекции окно фиксированного размера
Подколлекции: xs takeRight n	Создает коллекцию, состоящую из последних n элементов xs (или из каких-либо произвольных n элементов, если порядок следования элементов не определен).
xs dropRight n	Создает коллекцию, представленную всей остальной частью коллекции, за исключением xs takeRight n
Объединители: xs zip ys	Объединяет итерируемые пары соответствующих элементов из xs и ys.
xs zipAll (ys, x, y)	Объединяет итерируемые пары соответствующих элементов из xs и ys, при этом более короткая последовательность расширяется, чтобы соответствовать более длинной, за счет добавления элементов x или y.
xs.zipWithIndex	Объединяет итерируемые пары элементов из xs с их индексами
Сравнение xs sameElements ys	Возвращает результат проверки xs и ys на содержание одних и тех же элементов в одном и том же порядке

Зачем нужны и Traversable, и Iterable

А зачем над `Iterable` нужен еще один трейт `Traversable`? Разве нельзя выполнить все задуманное с помощью `iterator`? В чем смысл использования еще более абстрактного трейта, который определяет свои методы в понятиях `foreach`, а не `iterator`? Одной из причин использования `Traversable` является то, что иногда проще и эффективнее предоставить реализацию `foreach`, чем реализацию `iterator`. Рассмотрим простой пример. Предположим, что нужна иерархия классов для двоичных деревьев, на листьях которых находятся целочисленные элементы. Эту иерархию можно сконструировать следующим образом:

```
sealed abstract class Tree
```

```
case class Branch(left: Tree, right: Tree) extends
Tree
case class Node(elem: Int) extends Tree
```

А теперь предположим, что нужно обеспечить деревьям возможность обхода их элементов. Для этого пусть `Tree` станет наследником `Traversable[Int]` и определит следующий метод `foreach`:

```
sealed abstract class Tree extends
Traversable[Int] {
  def foreach[U](f: Int => U) = this match {
    case Node(elem) => f(elem)
    case Branch(l, r) => l foreach f; r foreach f
  }
}
```

Усилий понадобилось совсем немного, к тому же получен весьма эффективный код — обход элементов сбалансированного дерева занимает время, пропорциональное количеству элементов в дереве. Чтобы убедиться в этом, давайте представим, что у сбалансированного дерева с количеством листьев `N` будет `N - 1` внутренних узлов класса `Branch`. Следовательно, общее количество шагов по обходу дерева составит `N + N - 1`.

Теперь сравним это решение с приданием дереву итерируемости. Для этого пусть `Tree` станет наследником `Iterable[Int]` и определит следующий метод `iterator`:

```
sealed abstract class Tree extends Iterable[Int] {
  def iterator: Iterator[Int] = this match {
    case Node(elem) => Iterator.single(elem)
    case Branch(l, r) => l.iterator ++ r.iterator
  }
}
```

На первый взгляд все это не сложнее решения с использованием `foreach`. Но тут встает вопрос эффективности, связанный с реализацией в методе конкатенации итератора `++`. При каждом создании элемента таким объединяющим итератором, как `l.iterator ++ r.iterator`, к вычислению для получения правого итератора нужно идти окольным путем (либо `l.iterator`, либо `r.iterator`). В целом для получения листа сбалансированного дерева из `N` листьев требуется `log(N)` окольных путей. Следовательно, стоимость посещения всех элементов дерева поднимается с `2N` для обхода с помощью метода `foreach` до `N log (N)` для обхода с помощью метода `iterator`. Если у дерева 1 млн элементов, это означает около 2 млн шагов для `foreach` и около 20 млн шагов для `iterator`. Получается, что у решения с `foreach` явное преимущество.

Подкатегории Iterable

В иерархии наследования ниже `Iterable` находятся три трейта: `Seq`, `Set` и `Map`. Эти трейты объединяет то, что все они реализуют трейт `PartialFunction`[122] с методами `apply` и `isDefinedAt`. Но способы реализации `PartialFunction` у каждого трейта свои.

Для последовательностей `apply` применяется для позиционного индексирования, в котором элементы всегда нумеруются с нуля. То есть `Seq(1, 2, 3)(1) == 2`. Для наборов `apply` является проверкой на принадлежность. Например, `Set('a', 'b', 'c')('b') == true`, а `Set()('a') == false`. И наконец, для отображений `apply` становится средством выбора. Например, `Map('a' -> 1, 'b' -> 10, 'c' -> 100)('b') == 10`.

Более подробно каждый из этих видов коллекций будет рассмотрен в следующих трех разделах.

24.5. Трейты последовательностей Seq, IndexedSeq и LinearSeq

Трейт Seq является представителем последовательностей. Последовательностью называется разновидность итерируемой коллекции, у которой есть длина и все элементы которой имеют фиксированные индексные позиции, их отсчет начинается с нуля. Операции с последовательностями (сведены в табл. 24.3) разбиваются на следующие категории.

- **Операции индексирования и длины** apply, isDefinedAt, length, indices и lengthCompare. Для Seq операция apply означает индексирование, стало быть, последовательность типа Seq[T] является частично применяемой функцией, получающей Int-аргумент (индекс) и выдающей элемент последовательности типа T. Иными словами, Seq[T] расширяет PartialFunction[Int, T]. Элементы последовательности индексируются от нуля до длины последовательности length за вычетом единицы. Метод length, применяемый в отношении последовательностей, является псевдонимом общего для коллекций метода size. Метод lengthCompare позволяет сравнивать длины двух последовательностей, даже если одна из них имеет бесконечную длину.

- **Операции поиска индекса** indexOf, lastIndexOf, indexOfSlice, lastIndexOfSlice, indexWhere, lastIndexWhere, segmentLength и prefixLength, которые возвращают индекс элемента, равного заданному значению или соответствующего некоторому условию.

- **Операции добавления** +:, :+ и padTo, которые возвращают новые последовательности, получаемые путем добавления к началу или концу последовательности.

- **Операции обновления** updated и patch, которые возвращают новую последовательность, получаемую путем замены некоторых элементов исходной последовательности.

- **Операции сортировки** `sorted`, `sortWith` и `sortBy`, которые сортируют элементы последовательности в соответствии с различными критериями.

- **Операции реверсирования** `reverse`, `reverseIterator` и `reverseMap`, которые выдают или обрабатывают элементы последовательности в обратном порядке, с последнего до первого.

- **Операции сравнения** `startsWith`, `endsWith`, `contains`, `corresponds` и `containsSlice`, которые определяют соотношение двух последовательностей или ищут элемент в последовательности.

- **Операции над множествами** `intersect`, `diff`, `union` и `distinct`, которые выполняют над элементами двух последовательностей операции, подобные операциям с множествами, или удаляют дубликаты.

Таблица 24.3. Операции в трейте Seq

Что собой представляет	Чем занимается
Индексирования и длины: xs(i)	(или после раскрытия xs apply i) Возвращает элемент xs по индексу i.
xs isDefinedAt i	Проверяет, содержится ли i в xs.indices.
xs.length	Возвращает длину последовательности (то же самое, что и size).
xs.lengthCompare ys	Возвращает –1, если xs короче ys, +1, если первая последовательность длиннее второй, и 0, если они одинаковой длины. Работает, даже если одна из последовательностей бесконечна.
xs.indices	Возвращает диапазон индексов xs от 0 до xs.length - 1
Поиска индекса: xs indexOf x	Возвращает индекс первого элемента в xs, равного значению x (существует несколько вариантов).
xs lastIndexOf x	Возвращает индекс последнего элемента в xs, равного значению x (существует несколько вариантов).

xs indexOfSlice ys	Возвращает первый индекс xs при условии, что следующие друг за другом элементы, начинающиеся с элемента с этим индексом, составляют последовательность ys.
xs lastIndexOfSlice ys	Возвращает последний индекс xs при условии, что следующие друг за другом элементы, начинающиеся с элемента с этим индексом, составляют последовательность ys.
xs indexWhere p	Возвращает индекс первого элемента в xs, удовлетворяющего условию p (существует несколько вариантов).
xs segmentLength (p, i)	Возвращает длину самого длинного непрерывного сегмента элементов в xs, начинающегося с xs(i), который удовлетворяет условию p.
xs prefixLength p	Возвращает длину самого длинного префикса элементов в xs, все элементы которого удовлетворяют условию p
Добавления: x +: xs	Создает новую последовательность со значением x, добавленным в начало xs.
xs :+ x	Создает новую последовательность со значением x, добавленным в конец xs.
xs padTo (len, x)	Создает последовательность, получающуюся из добавления значения x к xs, до тех пор, пока длина не достигнет значения len
Обновления: xs patch (i, ys, r)	Создает последовательность, получающуюся заменой r элементов последовательности xs, начиная с i, элементами последовательности ys.
xs updated (i, x)	Создает копию xs, в которой элемент с индексом i заменяется значением x.
xs(i) = x	(или после раскрытия xs.update(i, x), которая доступна только для изменяемых последовательностей mutable.Seq) Изменяет значение элемента xs с индексом i на значение x
Что собой представляет	**Чем занимается**
Сортировки: xs.sorted	Создает новую последовательность путем сортировки элементов xs с использованием стандартного порядка следования элементов типа, хранящегося в xs.
xs sortWith lessThan	Создает новую последовательность путем сортировки элементов xs с использованием в качестве операции сравнения lessThan (меньше чем).
xs sortBy f	Создает новую последовательность путем сортировки элементов xs. Сравнение двух элементов выполняется путем применения к ним функции f и сравнения результатов
Реверсирования:	Создает последовательность из элементов xs, следующих в обратном

xs.reverse	порядке.
xs.reverseIterator	Создает итератор, выдающий все элементы xs в обратном порядке.
xs reverseMap f	Создает последовательность, получаемую путем применения функции f к элементам xs, стоящим в обратном порядке
Сравнения: xs startsWith ys	Проверяет, не начинается ли xs с последовательности ys (имеется несколько вариантов).
xs endsWith ys	Проверяет, не заканчивается ли xs последовательностью ys (имеется несколько вариантов).
xs contains x	Проверяет, имеется ли в xs элемент, равный x.
xs containsSlice ys	Проверяет, имеется ли в xs непрерывная последовательность, равная ys.
(xs corresponds ys)(p)	Проверяет, имеются ли в xs и ys элементы, удовлетворяющие бинарному предикату p
Операции над множествами: xs intersect ys	Выдает результат пересечения множеств, состоящих из последовательностей xs и ys, придерживаясь порядка следования элементов в xs.
xs diff ys	Выдает результат разности множеств, состоящих из последовательностей xs и ys, придерживаясь порядка следования элементов в xs.
xs union ys	Выдает результат объединения множеств (то же самое, что и xs ++ ys).
xs.distinct	Выдает часть последовательности xs, не содержащую продублированных элементов

Если последовательность изменяемая, для нее предлагается дополнительный метод добавления `update` с побочным эффектом, который позволяет элементам последовательности обновляться. Вспомним: в главе 3 уже говорилось, что синтаксис вида `seq(idx) = elem` является не более чем сокращением для выражения `seq.update(idx, elem)`. Обратите внимание на разницу между `update` и `updated`. Метод `update` изменяет элемент последовательности на месте и доступен только для изменяемых последовательностей. Метод `updated` доступен для всех последовательностей и всегда вместо изменения исходной возвращает новую последовательность.

У каждого трейта `Seq` имеется два подтрейта: `LinearSeq` и

IndexedSeq. Они не добавляют никаких новых операций, но каждый их них предлагает разные характеристики производительности. У линейной последовательности (linear sequence) имеются эффективные операции head и tail, а у индексированной последовательности — эффективные операции apply, length и (если последовательность изменяемая) update. В List зачастую используется линейная последовательность, как и в Stream. Представителями двух часто используемых индексированных последовательностей являются Array и ArrayBuffer. Класс Vector обеспечивает весьма интересный компромисс между индексированным и линейным доступом. У него имеются практически постоянные линейные издержки как на индексирование, так и на линейный доступ. Поэтому векторы служат хорошей основой для смешанных схем доступа, где используется как индексированный, так и линейный доступ. Более подробно векторы рассматриваются в разделе 24.8.

Буферы. Важной подкатегорией изменяемых последовательностей являются буферы. Они позволяют не только обновлять существующие элементы, но и вставлять и удалять элементы, а также с высокой эффективностью добавлять новые элементы в конец буфера. Принципиально новыми методами, поддерживаемыми буфером, являются += и ++= для добавления элементов к концу буфера, +=: и ++=: для добавления элементов в начало буфера, insert и insertAll для вставки элементов, а также remove и -= для удаления элементов. Все эти операции сведены в табл. 24.4.

Таблица 24.4. Операции в трейте Buffer

Что собой представляет	Чем занимается
Добавления: buf += x	Добавляет элемент x к буферу buf и возвращает в качестве результата сам buf.

buf += (x, y, z)	Добавляет к буферу указанные элементы.
buf ++= xs	Добавляет к буферу все элементы, имеющиеся в xs.
x +=: buf	Добавляет элемент x в начало буфера.
xs ++=: buf	Добавляет в начало буфера все элементы, имеющиеся в xs.
buf insert (i, x)	Вставляет элемент x в то место в буфере, на которое указывает индекс i.
buf insertAll (i, xs)	Вставляет все элементы, имеющиеся в xs, в то место в буфере, на которое указывает индекс i
Удаления: buf -= x	Удаляет из буфера элемент x.
buf remove i	Удаляет из буфера элемент с индексом i.
buf remove (i, n)	Удаляет из буфера n элементов, начиная с элемента с индексом i
buf trimStart n	Удаляет из буфера первые n элементов.
buf trimEnd n	Удаляет из буфера последние n элементов.
buf.clear()	Удаляет из буфера все элементы
Клонирование buf.clone	Создает новый буфер с теми же элементами, что и в buf

Наиболее часто используемыми являются две реализации буферов: `ListBuffer` и `ArrayBuffer`. Исходя из названий, `ListBuffer` базируется на классе `List` и поддерживает высокоэффективное преобразование своих элементов в список, а `ArrayBuffer` базируется на массиве и может быть быстро превращен в массив. Наметки реализации `ListBuffer` были показаны в разделе 22.2.

24.6. Наборы

Коллекции `Set` относятся к итерируемым `Iterable`-коллекциям, не содержащим повторяющихся элементов. Общие операции над наборами сведены в табл. 24.5, а в табл. 24.6 показаны операции для изменяемых наборов. Операции разбиты на следующие категории.

- **Проверки** `contains`, `apply` и `subsetOf`. Метод `contains`

показывает, содержит ли набор заданный элемент. Метод `apply` для набора является аналогом `contains`, поэтому `set(elem)` — это то же самое, что и `set contains elem`. Следовательно, наборы могут также использоваться в качестве тестовых функций, возвращающих `true` для содержащихся в них элементов, например:

```
scala> val fruit = Set("apple", "orange", "peach",
"banana")
fruit: scala.collection.immutable.Set[String] =
Set(apple, orange, peach, banana)

scala> fruit("peach")
res7: Boolean = true

scala> fruit("potato")
res8: Boolean = false
```

- **Добавления** `+` и `++`, которые добавляют к набору один и более элементов, выдавая в качестве результата новый набор.

- **Удаления** `-` и `--`, которые удаляют из набора один и более элементов, выдавая новый набор.

- **Операции над наборами** для объединения, выявления пересечений и разности наборов. Эти операции над наборами имеются в двух формах: текстовом и символьном. К текстовым формам относятся версии `intersect`, `union` и `diff`, а к символьным — `&`, `|` и `&~`. Оператор `++`, наследуемый `Set` из `Traversable`, может рассматриваться в качестве еще одного псевдонима `union` или `|`, за исключением того, что `++` получает `Traversable`-аргумент, а `union` и `|` получают наборы.

Таблица 24.5. Операции в трейте Set

Что собой представляет	Чем занимается
Проверки: xs contains x	Проверяет наличие x в качестве элемента xs.
xs(x)	Делает то же самое, что и xs contains x.
xs subsetOf ys	Проверяет, является ли xs поднабором ys
Продолжение^	

Таблица 24.5 (продолжение)

Что собой представляет	Чем занимается
Добавления: xs + x	Возвращает набор, содержащий все элементы xs и x.
xs + (x, y, z)	Возвращает набор, содержащий все элементы xs, а также заданные добавляемые элементы.
xs ++ ys	Возвращает набор, содержащий все элементы xs и ys
Удаления: xs – x	Возвращает набор, содержащий все элементы xs, за исключением x.
xs – (x, y, z)	Возвращает набор, содержащий все элементы xs, за исключением заданных элементов.
xs –– ys	Возвращает набор, содержащий все элементы xs, за исключением ys.
xs.empty	Возвращает пустой набор того же класса, что и xs
Бинарные операции: xs & ys	Возвращает набор пересечений xs и ys.
xs intersect ys	Возвращает то же самое, что и xs & ys.
xs | ys	Возвращает набор, объединяющий xs и ys.
xs union ys	Возвращает то же самое, что и xs | ys.
xs & ~ ys	Возвращает набор разности xs и ys.
xs diff ys	Возвращает то же самое, что и xs & ~ ys

У изменяемых наборов имеются методы, добавляющие, удаляющие и обновляющие элементы, которые сведены в

табл. 24.6.

Таблица 24.6. Операции в трейте mutable.Set

Что собой представляет	Чем занимается
Добавления: xs += x	В качестве побочного эффекта добавляет элемент x к набору xs и возвращает сам набор xs.
xs += (x, y, z)	В качестве побочного эффекта добавляет заданные элементы к набору xs и возвращает сам набор xs.
xs ++= ys	В качестве побочного эффекта добавляет все элементы в ys к набору xs и возвращает сам набор xs.
xs add x	Добавляет элемент x к xs и возвращает true, если x прежде не был в наборе, или false, если уже был
Удаления: xs −= x	В качестве побочного эффекта удаляет из набора xs элемент x и возвращает сам набор xs.
xs −= (x, y, z)	В качестве побочного эффекта удаляет из набора xs заданные элементы и возвращает сам набор xs.
xs −−= ys	В качестве побочного эффекта удаляет все элементы, находящиеся в ys, из набора xs и возвращает сам набор xs.
xs remove x	Удаляет элемент x из xs и возвращает true, если x прежде уже был в наборе, или false, если его прежде там не было.
Что собой представляет	**Чем занимается**
xs retain p	Сохраняет только те элементы в xs, которые удовлетворяют условию p.
xs.clear()	Удаляет из xs все элементы
Обновление xs(x) = b	(или после раскрытия xs.update(x, b)) Если аргумент b типа Boolean имеет значение true, добавляет x к xs, в противном случае удаляет x из xs
Клонирование xs.clone	Возвращает новый изменяемый набор с такими же элементами, как и в xs

Изменяемый набор, как и неизменяемый, предлагает для добавления элементов операции + и ++, а для удаления — операции - и --. Но для изменяемых наборов они используются гораздо реже, поскольку задействуют копирование набора. В качестве более эффективной альтернативы изменяемые наборы

предлагают методы обновления += и -=. Операция s += elem в качестве побочного эффекта добавляет elem к набору s и в качестве результата возвращает измененный набор. По аналогии с этим s -= elem удаляет элемент elem из набора и возвращает в качестве результата измененный набор. Кроме += и -=, есть также операции над несколькими элементами ++= и --=, которые добавляют или удаляют все элементы обходимой или итерируемой коллекции.

Выбор в качестве имен методов += и -= означает, что очень похожий код может работать как с изменяемыми, так и с неизменяемыми наборами. Рассмотрим сначала диалог с интерпретатором, в котором используется неизменяемый набор s:

```
scala> var s = Set(1, 2, 3)
s: scala.collection.immutable.Set[Int] = Set(1, 2, 3)

scala> s += 4; s -= 2

scala> s

res10: scala.collection.immutable.Set[Int] = Set(1, 3, 4)
```

В этом примере в отношении var-переменной типа immutable.Set используются методы += и -=. Согласно объяснениям, которые были даны на шаге 10 главы 3, инструкции вида s += 4 являются сокращенной формой записи для s = s + 4. Следовательно, в их выполнении участвует еще один метод +, применяемый в отношении набора s, а затем результат присваивается переменной s. А теперь рассмотрим аналогичную работу в интерпретаторе с изменяемым набором:

```
scala> val s = collection.mutable.Set(1, 2, 3)
```

```
s: scala.collection.mutable.Set[Int] = Set(1, 2,
3)

scala> s += 4
res11: s.type = Set(1, 2, 3, 4)

scala> s -= 2
res12: s.type = Set(1, 3, 4)
```

Конечный эффект очень похож на предыдущий диалог с интерпретатором: начинаем мы с набором `Set(1, 2, 3)`, а заканчиваем с набором `Set(1, 3, 4)`. Но даже при том что инструкции выглядят такими же, как и раньше, они выполняют несколько иные действия. Теперь инструкция `s += 4` вызывает метод `+=` в отношении значения `s`, представляющего собой изменяемый набор, выполняя изменения на месте. Аналогично этому инструкция `s -= 2` теперь вызывает в отношении этого же набора метод `-=`.

Сравнение этих двух диалогов позволяет выявить весьма важный принцип. Зачастую можно заменить изменяемую коллекцию, хранящуюся в `val`-переменной, неизменяемой коллекцией, хранящейся в `var`-переменной, и наоборот. Это работает по крайней мере до тех пор, пока нет псевдонимов ссылок на коллекцию, позволяющих заметить, обновилась она на месте или была создана новая коллекция.

Изменяемые наборы также предоставляют в качестве вариантов `+=` и `-=` методы `add` и `remove`. Разница в том, что методы `add` и `remove` возвращают булев результат, показывающий, возымела операция эффект над набором или нет.

В текущей исходной реализации изменяемого набора для хранения элементов набора применяется хеш-таблица. В исходной реализации неизменяемых наборов используется представление, которое адаптируется к количеству элементов набора. Пустой

набор представляется в виде простого синглтон-объекта. Наборы размером до четырех элементов представляются в виде одиночного объекта, сохраняющего все элементы в полях. Все неизменяемые наборы, имеющие другие размеры, реализуются в виде хеш-извлечений[123].

Последствия применения таких вариантов представления заключаются в том, что для наборов небольших размеров с количеством элементов, не превышающим четырех, неизменяемые наборы получаются более компактными и более эффективными в работе, чем изменяемые. Поэтому, если предполагается, что набор будет небольшим, попробуйте сделать его неизменяемым.

24.7. Отображения

Отображения типа `Map` представляют собой итерируемые `Iterable`-коллекции, состоящие из пар ключей и значений, которые также называются сопоставлениями или ассоциациями. Как говорилось в разделе 21.4, имеющийся в Scala класс `Predef` предлагает подразумеваемое преобразование, позволяющее использовать запись вида `ключ -> значение` в качестве альтернативы синтаксиса для пары вида `(ключ, значение)`. Таким образом, выражение `Map("x" -> 24, "y" -> 25, "z" -> 26)` означает абсолютно то же самое, что и выражение `Map(("x", 24), ("y", 25), ("z", 26))`, но читается легче.

Основные операции над отображениями, сведенные в табл. 24.7, похожи на аналогичные операции над наборами. Изменяемые отображения, кроме того, поддерживают операции, показанные в табл. 24.8. Операции над отображениями разбиваются на следующие категории.

- **Поиска** `apply`, `get`, `getOrElse`, `contains` и `isDefinedAt`. Эти операции превращают отображения в частично применяемые

функции от ключей к значениям.

Основной метод поиска для отображений выглядит следующим образом:

```
def get(key): Option[Value]
```

Операция `m get key` проверяет, содержит ли отображение ассоциацию для заданного ключа. Если такая ассоциация имеется, она возвращает значение ассоциации в виде объекта типа `Some`. Если такой ключ в отображении не определен, `get` возвращает `None`. В отображениях также определяется метод `apply`, возвращающий значение, непосредственно ассоциированное с заданным ключом, без его инкапсуляции в `Option`. Если ключ в отображении не определен, выдается исключение.

- **Добавления и обновления** `+`, `++` и `updated`, позволяющие добавлять к отображению новые связки или изменять уже существующие.

- **Удаления** `-` и `--`, позволяющие удалять связки из отображения.

- **Операции создания подколлекций** `keys`, `keySet`, `keysIterator`, `valuesIterator` и `values`, возвращающие по отдельности ключи и значения отображений в различных формах.

- **Преобразований** `filterKeys` и `mapValues`, создающие новое отображение путем фильтрации и преобразования связок существующего отображения.

Таблица 24.7. Операции в трейте Map

Что собой представляет	Чем занимается

Поиски: ms get k	Возвращает как вариант значение, связанное с ключом k в отображении ms, или None, если ключ не найден..
ms(k)	(или после раскрытия ms apply k) Возвращает значение, связанное с ключом k в отображении ms, или выдает исключение, если ключ не найден.
ms getOrElse (k, d)	Возвращает значение, связанное с ключом k в отображении ms, или значение по умолчанию d, если ключ не найден.
ms contains k	Проверяет, содержится ли в ms отображение для ключа k.
ms isDefinedAt k	Возвращает то же самое, что и contains
Добавления и обновления: ms + (k –> v)	Возвращает отображение, содержащее все, что входит в ms, а также отображение k –> v, то есть ключа k на значение v.
ms + (k –> v, l –> w)	Возвращает отображение, содержащее все, что хранится в ms, а также указанные пары «ключ — значение».
ms ++ kvs	Возвращает отображение, содержащее все, что входит в ms, а также все пары «ключ — значение» в kvs.
ms updated (k, v)	Возвращает то же самое, что и ms + (k –> v)
Удаления: ms – k	Возвращает отображение, содержащее все, что содержится в ms, за исключением любого отображения ключа k.
Продолжение^	

Таблица 24.7 (продолжение)

Что собой представляет	Чем занимается
ms – (k, l, m)	Возвращает отображение, содержащее все, что хранится в ms, за исключением любых отображений указанных ключей.
ms – ks	Возвращает отображение, содержащее все, что содержится в ms, за исключением любых отображений с ключом в ks
Создание подколлекций: ms.keys	Возвращает обходимую коллекцию, содержащую каждый ключ, имеющийся в ms.
ms.keySet	Возвращает набор, содержащий каждый ключ, имеющийся в ms.
ms.keysIterator	Возвращает итератор, выдающий каждый ключ, имеющийся в ms.
	Возвращает обходимую коллекцию, содержащую каждое значение,

ms.values	связанное с ключом в ms.
ms.valuesIterator	Возвращает итератор, выдающий каждое значение, связанное с ключом в ms
Преобразования: ms filterKeys p	Возвращает представление отображения, содержащее только те отображения, имеющиеся в ms, в которых ключ удовлетворяет условию p.
ms mapValues f	Возвращает представление отображения, получающееся в результате применения функции f к каждому значению, связанному с ключом в ms

Таблица 24.8. Операции в трейте mutable.Map

Что собой представляет	Чем занимается
Добавления и обновления: ms(k) = v	(или после раскрытия ms.update(k, v)) Добавляет в качестве побочного эффекта отображение ключа k на значение v к отображению ms, перезаписывая все ранее имевшиеся отображения k.
ms += (k –> v)	Добавляет в качестве побочного эффекта отображение ключа k на значение v к отображению ms и возвращает само отображение ms.
ms += (k –> v, l –> w)	Добавляет в качестве побочного эффекта заданное отображение к отображению ms и возвращает само отображение ms.
ms ++= kvs	Добавляет в качестве побочного эффекта все отображения, имеющиеся в kvs, к ms и возвращает само отображение ms.
ms put (k, v)	Добавляет к ms отображение ключа k на значение v и возвращает как вариант любое значение, ранее связанное с k.
ms getOrElseUpdate (k, d)	Если ключ k определен в отображении ms, возвращает связанное с ним значение. В противном случае обновляет ms отображением k –> d и возвращает d
Удаления: ms –= k	Удаляет в качестве побочного эффекта отображение с ключом kиз ms и возвращает само отображение ms.
Что собой представляет	**Чем занимается**
ms –= (k, l, m)	Удаляет в качестве побочного эффекта отображения с заданными ключами из ms и возвращает само отображение ms.
ms ––= ks	Удаляет в качестве побочного эффекта все отображения из ms с ключами, имеющимися в ks, и возвращает само отображение ms.
	Удаляет все отображения с ключом k из ms и возвращает как вариант

ms remove k	любое значение, ранее связанное с k.
ms retain p	Сохраняет в ms только те отображения, у которых имеется ключ, удовлетворяющий условию p.
ms.clear()	Удаляет из ms все отображения
Преобразование и клонирование: ms transform f	Выполняет преобразование всех связанных значений в отображении ms с помощью функции f
ms.clone	Возвращает новое изменяемое отображение с такими же отображениями, как и в ms

Операции добавления и удаления для отображений являются зеркальными отражениями таких же операций для наборов. Как и наборы, изменяемые отображения поддерживают неразрушающие операции добавления +, - и updated, но они используются менее часто, поскольку влекут за собой копирование изменяемого отображения. Вместо этого изменяемое отображение m обычно обновляется на месте, с использованием двух вариантов: m(ключ) = значение или m +=(ключ -> значение). Есть также вариант m put(ключ, значение), который возвращает Option-значение, содержащее то, что прежде ассоциировалось с ключом, или None, если ранее такой ключ в отображении отсутствовал.

Метод getOrElseUpdate пригодится для обращения к отображениям там, где они действуют в качестве кэш-областей памяти. Скажем, у вас имеется весьма затратное вычисление, запускаемое путем вызова функции f:

```
scala> def f(x: String) = {
                    println("taking my time.");
Thread.sleep(100)
       x.reverse }
f: (x: String)String
```

Далее предположим, что у f имеется побочный эффект, поэтому ее повторный вызов с тем же самым аргументом всегда будет

выдавать тот же самый результат. В таком случае можно сберечь время, сохранив ранее вычисленные связки аргумента и результата выполнения `f` в отображении, и рассчитывать результат выполнения `f`, только если результат для аргумента не был найден в отображении. Можно назвать отображение областью кэш-памяти для вычислений функции `f`:

```
scala> val cache = collection.mutable.Map[String, String]()
cache: scala.collection.mutable.Map[String,String] = Map()
```

Теперь можно создать более эффективную кэшированную версию функции `f`:

```
scala> def cachedF(s: String) = cache.getOrElseUpdate(s, f(s))
cachedF: (s: String)String
scala> cachedF("abc")
taking my time.
res16: String = cba

scala> cachedF("abc")
res17: String = cba
```

Обратите внимание на то, что второй аргумент `getOrElseUpdate` является аргументом до востребования, следовательно, показанное ранее вычисление `f("abc")` выполняется только в том случае, если методу `getOrElseUpdate` потребуется значение его второго аргумента, что происходит именно тогда, когда его первый аргумент не найден в кэширующем отображении. Нужно будет также непосредственным образом реализовать `cachedF`, используя только основные операции с отображениями, но для этого понадобится

дополнительный код:

```
def cachedF(arg: String) = cache get arg match {
  case Some(result) => result
  case None =>
    val result = f(arg)
    cache(arg) = result
    result
}
```

24.8. Конкретные классы неизменяемых коллекций

В Scala на выбор предлагается множество конкретных классов неизменяемых коллекций. Они отличаются особенностями своей реализации (поскольку представляют собой отображения, наборы, последовательности), возможностью иметь бесконечное количество элементов и скоростью выполнения различных операций над ними. Начнем с обзора наиболее востребованных типов неизменяемых коллекций.

Списки

Списки относятся к конечным неизменяемым последовательностям. Они обеспечивают постоянное время доступа к своим первым элементам, а также ко всей остальной части списка, и у них постоянное время выполнения операций `cons` для добавления нового элемента к началу списка. Многие другие операции имеют линейную зависимость времени выполнения от длины списка. Более подробно списки рассматриваются в главах 16 и 22.

Потоки

Потоки похожи на списки, за исключением того, что в отношении

их элементов применяется «ленивое» вычисление. Поэтому потоки могут быть бесконечно длинными. Вычисляться будут только запрошенные элементы. Во всем остальном потоки имеют такие же характеристики производительности, что и списки.

В то время как списки составляются с помощью оператора `::`, потоки формируются с помощью похожего оператора `#::`. Пример потока, содержащего целые числа 1, 2 и 3, выглядит следующим образом:

```
scala> val str = 1 #:: 2 #:: 3 #:: Stream.empty
str:    scala.collection.immutable.Stream[Int]    =
Stream(1, ?)
```

Головой этого потока является 1, а хвостом — 2 и 3. Но хвост здесь не выводится, поскольку он еще не вычислен! Для потоков следует применять «ленивое» вычисление, и метод `toString`, вызванный в отношении потока, заботится о том, чтобы не навязывались никакие лишние вычисления.

Далее показан более сложный пример. В нем вычисляется поток, содержащий последовательность чисел Фибоначчи с двумя заданными числами. Каждый элемент последовательности Фибоначчи является суммой двух предыдущих элементов в серии:

```
scala> def fibFrom(a: Int, b: Int): Stream[Int] =
         a #:: fibFrom(b, a + b)
fibFrom: (a: Int, b: Int)Stream[Int]
```

Эта функция кажется обманчиво простой. Понятно, что первым элементом последовательности является `a`, за ним стоит последовательность Фибоначчи, начинающаяся с `b`, затем — элемент со значением `a + b`. Самое сложное здесь — вычислить данную последовательность, не вызвав бесконечную рекурсию. Если функция вместо `#::` использует оператор `::`, то каждый вызов функции будет приводить к еще одному вызову, что

выльется в бесконечную рекурсию. Но, поскольку применяется оператор #::, правая часть выражения не вычисляется до тех пор, пока не будет востребована.

Вот как выглядят первые несколько элементов последовательности Фибоначчи, начинающейся с двух заданных чисел:

```
scala> val fibs = fibFrom(1, 1).take(7)
fibs: scala.collection.immutable.Stream[Int] = Stream(1, ?)

scala> fibs.toList
res23: List[Int] = List(1, 1, 2, 3, 5, 8, 13)
```

Векторы

Списки высокоэффективны там, где алгоритм обработки занимается только их головными элементами. Обращение к голове, добавление к началу списка или удаление из него элемента занимает постоянное время, а вот обращение к последующим элементам или их обновление занимает время, пропорциональное глубине списка.

Векторы относятся к типу коллекций, предоставляющему эффективный доступ к элементам, следующим после головы. Доступ к любому элементу вектора занимает, как выяснится чуть позже, практически постоянное время. Оно больше времени доступа к голове списка или чтения элемента массива, но все же постоянно. В результате алгоритмы, использующие векторы, не должны ограничиваться доступом только к голове последовательности. Они могут обращаться к элементам и изменять их в произвольном месте и гораздо удобнее в написании.

Создаются и изменяются векторы практически так же, как и любые другие последовательности:

```
scala> val vec = scala.collection.immutable.Vector.empty
vec: scala.collection.immutable.Vector[Nothing] = Vector()

scala> val vec2 = vec :+ 1 :+ 2
vec2: scala.collection.immutable.Vector[Int] = Vector(1, 2)

scala> val vec3 = 100 +: vec2
vec3: scala.collection.immutable.Vector[Int]
  = Vector(100, 1, 2)

scala> vec3(0)
res24: Int = 100
```

Для представления векторов используются широкие неглубокие деревья. Каждый узел дерева содержит до 32 элементов вектора или состоит не более чем из 32 других узлов дерева. Векторы, содержащие до 32 элементов, могут быть представлены одним узлом. Векторы с количеством элементов до $32 \times 32 = 1024$ могут быть представлены в виде одного направления. Двух переходов от корня дерева к конечному элементу достаточно для векторов, имеющих до 2^{15} элементов, трех — для векторов с количеством элементов, равным 2^{20}, четырех — для векторов с количеством элементов, равным 2^{25}, а пяти — для векторов с количеством элементов, равным 2^{30}. Следовательно, для всех векторов разумного размера выбор элемента требует до пяти обычных выборов массивов. Именно поэтому для векторов было выбрано определение «практически постоянное время».

Векторы являются неизменяемыми, следовательно, внести в элемент вектора изменение на месте невозможно. Но с помощью метода `updated` реально создать новый вектор, отличающийся от

заданного только одним элементом:

```
scala> val vec = Vector(1, 2, 3)
vec:    scala.collection.immutable.Vector[Int]    =
Vector(1, 2, 3)

scala> vec updated (2, 4)
res25:   scala.collection.immutable.Vector[Int]   =
Vector(1, 2, 4)

scala> vec
res26:   scala.collection.immutable.Vector[Int]   =
Vector(1, 2, 3)
```

Последняя строка данного кода показывает, что вызов метода `updated` никак не повлиял на исходный вектор `vec`. Функциональное обновление векторов также занимает практически постоянное время. Обновление элемента в середине вектора может быть произведено копированием узла, содержащего элемент, и каждого указывающего на него узла, начиная с корня дерева. Это означает, что функциональное обновление создает от одного до пяти узлов, каждый из которых содержит до 32 элементов или поддеревьев. Конечно, это гораздо затратнее обновления на месте в изменяемом массиве, но все же дешевле копирования всего вектора.

Поскольку векторы сохраняют разумный баланс между быстрым произвольным выбором и быстрыми произвольными функциональными обновлениями, в настоящий момент они представляют исходную реализацию неизменяемых проиндексированных последовательностей:

```
scala> collection.immutable.IndexedSeq(1, 2, 3)
res27: scala.collection.immutable.IndexedSeq[Int]
  = Vector(1, 2, 3)
```

Неизменяемые стеки

Если нужна последовательность, работающая по принципу «последним пришел, первым вышел», можно воспользоваться значением типа `Stack`. Помещается элемент в стек с помощью метода `push`, извлекается из стека с помощью метода `pop`, а заглянуть на вершину стека без удаления находящегося там элемента можно с помощью метода `top`. На все эти операции затрачивается постоянное время.

Вот как выглядят несколько простых операций, выполняемых над стеком:

```
scala> val stack = scala.collection.immutable.Stack.empty
stack: scala.collection.immutable.Stack[Nothing] = Stack()

scala> val hasOne = stack.push(1)
hasOne: scala.collection.immutable.Stack[Int] = Stack(1)

scala> stack
res28: scala.collection.immutable.Stack[Nothing] = Stack()

scala> hasOne.top
res29: Int = 1

scala> hasOne.pop
res30: scala.collection.immutable.Stack[Int] = Stack()
```

Неизменяемые стеки используются в программах на Scala довольно редко, поскольку их функциональные возможности могут

реализовывать списки: применение метода push в отношении неизменяемого стека — по сути, то же самое, что применение метода :: в отношении списка, а применение метода pop в отношении стека аналогично применению метода tail в отношении списка.

Неизменяемые очереди

Очереди очень похожи на стек, за исключением того, что в них используется принцип «первым пришел, первым вышел», а не принцип «последним пришел, первым вышел». Упрощенная реализация неизменяемых очередей рассматривалась в главе 19. Создать пустую неизменяемую очередь можно следующим образом:

```
scala> val empty = scala.collection.immutable.Queue[Int]()
empty: scala.collection.immutable.Queue[Int] = Queue()
```

Добавить к неизменяемой очереди элемент можно с помощью метода enqueue:

```
scala> val has1 = empty.enqueue(1)
has1: scala.collection.immutable.Queue[Int] = Queue(1)
```

Чтобы добавить к очереди сразу несколько элементов, следует вызвать enqueue с коллекцией в качестве его аргументов:

```
scala> val has123 = has1.enqueue(List(2, 3))
has123: scala.collection.immutable.Queue[Int] = Queue(1, 2, 3)
```

Чтобы удалить элемент из головы очереди, нужно воспользоваться методом dequeue:

```
scala> val (element, has23) = has123.dequeue
element: Int = 1
has23:   scala.collection.immutable.Queue[Int]   =
Queue(2, 3)
```

Обратите внимание на то, что dequeue возвращает пару, состоящую из удаленного элемента и оставшейся части очереди.

Диапазоны

Диапазон представляет собой упорядоченную последовательность целых чисел, отстающих друг от друга на одинаковую величину. Например, «1, 2, 3» является диапазоном точно так же, как и «5, 8, 11, 14». Чтобы создать в Scala диапазон, следует воспользоваться предопределенными методами to и by. Рассмотрим несколько примеров:

```
scala> 1 to 3
res31: scala.collection.immutable.Range.Inclusive
  = Range(1, 2, 3)

scala> 5 to 14 by 3
res32: scala.collection.immutable.Range = Range(5,
8, 11, 14)
```

Если нужно создать диапазон, исключая его верхний предел, следует вместо метода to воспользоваться весьма удобным методом until:

```
scala> 1 until 3
res33: scala.collection.immutable.Range = Range(1,
2)
```

Диапазоны представлены постоянным объемом памяти, поскольку могут быть определены всего тремя числами: их

началом, их концом и значением шага. Благодаря такому представлению большинство операций над диапазонами выполняется очень быстро.

Хеш-извлечения

Хеш-извлечения (hash-tries[124]) являются стандартным способом эффективной реализации неизменяемых наборов и отображений. Их представление похоже на векторы тем, что в них также используются деревья, где у каждого узла имеется 32 элемента или 32 поддерева, но выбор осуществляется на основе хеш-кода. Например, самые младшие пять разрядов хеш-кода ключа используются для поиска заданного ключа в отображении для выбора первого поддерева, следующие пять — для выбора следующего поддерева и т. д. Процесс выбора прекращается, как только у всех элементов, хранящихся в узле, будут хеш-коды, отличающиеся друг от друга во всех выбранных на данный момент разрядах. Таким образом, бывает, что в этом процессе используются не все разряды хеш-кода.

Хеш-извлечения позволяют достичь хорошего баланса между достаточно быстрым поиском и достаточно эффективными функциональными вставкой (+) и удалением (-). Именно поэтому они положены в основу имеющихся в Scala исходных реализаций неизменяемых отображений и наборов. Фактически для неизменяемых наборов и отображений, содержащих менее пяти элементов, в Scala предусматривается дальнейшая оптимизация. Наборы и отображения, включающие от одного до четырех элементов, хранятся в виде отдельных объектов, содержащих эти элементы (или в случае с отображениями — пары «ключ — значение») в виде полей. Пустой изменяемый набор и пустое изменяемое отображение во всех случаях являются синглтон-объектами из-за отсутствия необходимости дублировать для них место хранения, поскольку пустые неизменяемые набор или отображение всегда будут оставаться пустыми.

Красно-черные деревья

Формой сбалансированных двоичных деревьев являются красно-черные деревья, где одни узлы обозначены как красные, а другие — как черные. Операции над ними, как и над любыми другими сбалансированными двоичными деревьями, гарантированно завершаются за время, находящееся в экспоненциальной зависимости от размера дерева.

В Scala имеются наборы и отображения, во внутренней реализации которых используется красно-черное дерево. Доступ к ним осуществляется по именам `TreeSet` и `TreeMap`:

```
scala> val set = collection.immutable.TreeSet.empty[Int]
set: scala.collection.immutable.TreeSet[Int] = TreeSet()

scala> set + 1 + 3 + 3
res34: scala.collection.immutable.TreeSet[Int] = TreeSet(1, 3)
```

Красно-черные деревья в Scala являются также стандартным механизмом реализации `SortedSet`, поскольку они предоставляют весьма эффективный итератор, возвращающий все элементы набора в отсортированном виде.

Неизменяемые наборы битов

Набор битов представляет коллекцию небольших целых чисел в виде битов большого целого числа. Например, набор битов, содержащий 3, 2 и 0, будет представлен в двоичном виде как целое число 1101, которое в десятичном виде является числом 13.

Во внутреннем представлении наборы битов используют массив из 64-разрядных `Long`-значений. Первое `Long`-значение в массиве предназначено для целых чисел от 0 до 63, второе — от 64 до 127 и

т. д. Таким образом, до тех пор, пока самое большое целое число в наборе не превышает нескольких сотен, наборы битов очень компактны.

Операции над наборами битов выполняются очень быстро. Проверка на присутствие занимает постоянное время. Добавление записи к набору занимает время, пропорциональное количеству Long-значений в массиве набора битов, которых обычно немного. Некоторые примеры использования набора битов выглядят следующим образом:

```
scala> val bits = scala.collection.immutable.BitSet.empty
bits: scala.collection.immutable.BitSet = BitSet()

scala> val moreBits = bits + 3 + 4 + 4
moreBits: scala.collection.immutable.BitSet = BitSet(3, 4)

scala> moreBits(3)
res35: Boolean = true

scala> moreBits(0)
res36: Boolean = false
```

Списочные отображения

Списочное отображение представляет собой отображение в виде связанного списка пар «ключ — значение». Как правило, операции над списочными отображениями могут потребовать перебора всего списка. В связи с этим операции над списочным отображением занимают время, пропорциональное размеру отображения. Списочные отображения не нашли в Scala широкого применения, поскольку работа со стандартными неизменяемыми отображениями почти всегда выполняется быстрее. Единственно

возможное положительное отличие наблюдается в том случае, если по какой-то причине отображение сконструировано так, что первые элементы списка выбираются гораздо чаще других элементов.

```
scala> val map = collection.immutable.ListMap(
         1 -> "one", 2 -> "two")
map:
scala.collection.immutable.ListMap[Int,String]   =
Map(1
-> one, 2 -> two)

scala> map(2)
res37: String = "two"
```

24.9. Конкретные классы изменяемых коллекций

После изучения наиболее востребованных из имеющихся в стандартной библиотеке Scala классов неизменяемых коллекций рассмотрим классы изменяемых коллекций.

Буферы массивов

Буферы массивов уже встречались в разделе 17.1. В буфере массива содержатся массив и его размер. Большинство операций над буферами массивов выполняется с такой же скоростью, что и над массивами, поскольку эти операции просто обращаются к обрабатываемому массиву и вносят в него изменения. Кроме того, у буфера массива могут быть данные, весьма эффективно добавляемые к его концу. Добавление записей к буферу массива при большом количестве операций занимает время, прямо пропорциональное количеству этих операций, то есть на добавление записи уходит амортизированно постоянное время (amortized constant time). Из этого следует, что буферы массивов

хорошо подходят для эффективного создания больших коллекций, когда новые записи всегда добавляются к концу. Вот как выглядят некоторые примеры их применения:

```
scala> val buf = collection.mutable.ArrayBuffer.empty[Int]
buf: scala.collection.mutable.ArrayBuffer[Int]
  = ArrayBuffer()

scala> buf += 1
res38: buf.type = ArrayBuffer(1)

scala> buf += 10
res39: buf.type = ArrayBuffer(1, 10)

scala> buf.toArray
res40: Array[Int] = Array(1, 10)
```

Списочные буферы

В разделе 17.1 встречались также списочные буферы. Списочный буфер похож на буфер массива, за исключением того, что внутри него используется не массив, а связанный список. Если сразу после создания буфер предполагается превращать в список, лучше воспользоваться списочным буфером, а не буфером массива. Вот как выглядит пример этого[125]:

```
scala> val buf = collection.mutable.ListBuffer.empty[Int]
buf: scala.collection.mutable.ListBuffer[Int]
  = ListBuffer()

scala> buf += 1
res41: buf.type = ListBuffer(1)
```

```
scala> buf += 10
res42: buf.type = ListBuffer(1, 10)
scala> buf.toList
res43: List[Int] = List(1, 10)
```

Построители строк

По аналогии с тем, что буфер массива используется для создания массивов, а списочный буфер — для создания списков, построитель строк применяется для создания строк. Построители строк используются настолько часто, что они заранее импортируются в исходное пространство имен. Для их создания применяется простое выражение `new StringBuilder`:

```
scala> val buf = new StringBuilder
buf: StringBuilder =

scala> buf += 'a'
res44: buf.type = a

scala> buf ++= "bcdef"
res45: buf.type = abcdef

scala> buf.toString
res46: String = abcdef
```

Связанные списки

Связанные списки представляют собой изменяемые последовательности, состоящие из узлов, связанных со следующими за ними указателями. В большинстве языков для пустого связанного списка было бы выбрано значение `null`. Но для коллекций Scala оно не подходит, поскольку даже пустые

последовательности должны поддерживать все методы работы с последовательностями. В частности, выражение `LinkedList.empty.isEmpty` должно возвращать `true`, а не выдавать исключение `NullPointerException`. Вместо этого у пустых связанных списков есть собственный способ кодирования: их поле `next` указывает на сам узел.

Как и их неизменяемые сородичи, связанные списки лучше всего обрабатывать последовательно. Вдобавок ко всему связанные списки облегчают вставку элемента или привязку списка к другому связанному списку.

Двунаправленные связанные списки

Объекты типа `DoubleLinkedList` похожи на однонаправленные связанные списки, рассмотренные в предыдущем подразделе, за исключением того, что, кроме поля `next`, у них имеется еще одно изменяемое поле, `prev`, указывающее на элемент, расположенный перед текущим узлом. Основное преимущество наличия этой дополнительной ссылки заключается в существенном ускорении удаления элементов.

Изменяемые списки

Объект типа `MutableList` состоит из однонаправленного связанного списка в сочетании с указателем, ссылающимся на пустой конечный узел этого списка. Тем самым время выполнения операции дополнения списка становится постоянным, поскольку этот указатель позволяет избежать перемещения по списку в поиске его конечного узла. В Scala `MutableList` является в настоящее время стандартной реализацией `mutable.LinearSeq`.

Очереди

Наряду с неизменяемыми очередями в Scala имеются и

изменяемые очереди. Используются они так же, как и неизменяемые, но для добавления элементов вместо enqueue применяются операторы += и ++=. Кроме того, метод dequeue будет просто удалять из изменяемой очереди головной элемент и возвращать его. Примеры использования:

```
scala> val queue = new scala.collection.mutable.Queue[String]
queue: scala.collection.mutable.Queue[String] = Queue()

scala> queue += "a"
res47: queue.type = Queue(a)

scala> queue ++= List("b", "c")
res48: queue.type = Queue(a, b, c)

scala> queue
res49: scala.collection.mutable.Queue[String] = Queue(a, b, c)

scala> queue.dequeue
res50: String = a

scala> queue
res51: scala.collection.mutable.Queue[String] = Queue(b, c)
```

Последовательности, применяющие массив

Последовательности, применяющие массив, представляют собой изменяемые последовательности фиксированного размера, внутри которых элементы хранятся в данных типа Array[AnyRef]. В Scala

они реализованы в классе ArraySeq.

Обычно тип ArraySeq применяется, если нужно получить такие же характеристики производительности, что и у массива, но вдобавок создать обобщенные экземпляры последовательностей, тип элементов которых неизвестен, и не имеется ClassTag для предоставления его в ходе выполнения программы. Все эти вопросы применительно к массивам будут рассмотрены в разделе 24.10.

Стеки

Неизменяемые стеки уже описывались в этой книге. Но есть еще их изменяемая версия. Она работает точно так же, как и неизменяемая, за исключением того, что изменения производятся на месте. Примеры использования:

```
scala> val stack = new scala.collection.mutable.Stack[Int]
stack: scala.collection.mutable.Stack[Int] = Stack()

scala> stack.push(1)
res52: stack.type = Stack(1)

scala> stack
res53: scala.collection.mutable.Stack[Int] = Stack(1)

scala> stack.push(2)
res54: stack.type = Stack(2, 1)

scala> stack
res55: scala.collection.mutable.Stack[Int] = Stack(2, 1)
```

```
scala> stack.top
res56: Int = 2

scala> stack
res57:    scala.collection.mutable.Stack[Int]    =
Stack(2, 1)

scala> stack.pop
res58: Int = 2

scala> stack
res59:    scala.collection.mutable.Stack[Int]    =
Stack(1)
```

Стеки, использующие массивы

Стек `ArrayStack` является альтернативой изменяемому стеку, основанной на использовании типа `Array`. Размер такого стека будет изменяться по мере необходимости. Он обеспечивает возможность быстрого индексирования и, как правило, при выполнении большинства операций чуть более эффективен, чем обычный изменяемый стек.

Хеш-таблицы

Хеш-таблица сохраняет свои элементы в образующем ее массиве, помещая каждую запись в позицию в массиве, определяемую хеш-кодом этой записи. На добавление элемента к хеш-таблице всегда уходит одно и то же время, если только в массиве нет еще одного элемента с точно таким же хеш-кодом. Поэтому до тех пор, пока помещенные в хеш-таблицу элементы имеют хорошее распределение хеш-кодов, работа с ней выполняется довольно быстро. По этой причине исходные типы изменяемых

отображений и наборов в Scala основаны на применении хеш-таблиц.

Хеш-наборы и хеш-отображения используются точно так же, как и любые другие наборы или отображения. Вот как выглядят некоторые примеры их использования:

```
scala> val map = collection.mutable.HashMap.empty[Int,String]
map: scala.collection.mutable.HashMap[Int,String] = Map()

scala> map += (1 -> "make a web site")
res60: map.type = Map(1 -> make a web site)

scala> map += (3 -> "profit!")
res61: map.type = Map(1 -> make a web site, 3 -> profit!)

scala> map(1)
res62: String = make a web site

scala> map contains 2
res63: Boolean = false
```

Конкретный порядок обхода элементов хеш-таблицы не гарантируется. Выполняется простой обход элементов массива, на котором основана хеш-таблица, в порядке расположения его элементов. Для получения гарантированного порядка обхода следует воспользоваться связанным хеш-отображением или набором, который похож на обычные хеш-отображение или набор, за исключением того, что в него включен связанный список элементов в порядке их добавления. Обход элементов такой коллекции всегда выполняется в том же порядке, в котором они изначально к ней добавлялись.

Слабые хеш-отображения

Слабое хеш-отображение представляет собой особую разновидность хеш-отображения, в которой сборщик мусора не следует по ссылкам от отображения к хранящимся в нем ключам. Это означает, что ключ и связанное с ним значение будут убраны из отображения, если на этот ключ нет другой ссылки. Слабые хеш-отображения используются для решения таких задач, как кэширование, когда нужно повторно использовать результат затратной функции, если функция вновь вызывается в отношении того же самого ключа. Если ключи и результаты функции хранятся в обычном хеш-отображении, оно может бесконечно разрастись и никакие ключи никогда не станут мусором. Этой проблемы удается избежать с помощью слабого хеш-отображения. Как только объект ключа становится недоступным, связанная с ним запись из такого отображения удаляется. Слабые хеш-отображения реализованы в Scala в виде оболочки положенной в их основу Java-реализации `java.util.WeakHashMap`.

Совместно используемые отображения

К совместно используемому отображению могут обращаться сразу несколько потоков. В дополнение к обычным операциям с `Map`-типами в этом отображении имеются атомарные операции (табл. 24.9).

Таблица 24.9. Операции в трейте ConcurrentMap

Что собой представляет	Чем занимается
m putIfAbsent(k, v)	Добавляет связку «ключ —значение» k -> m, кроме тех случаев, когда k уже определен в m
m remove (k, v)	Удаляет запись для ключа k, если он в данный момент отображен на значение v
m replace (k, old, new)	Заменяет значение, связанное с k новым значением new, если ранее с ключом было связано значение old

m replace (k, v)	Заменяет значение, связанное с k, значением v, если ранее этот ключ был связан с каким-либо значением

ConcurrentMap является трейтом, который находится в библиотеке коллекций Scala. Сейчас существует только одна его Java-реализация в java.util.concurrent.ConcurrentMap, которая может быть автоматически преобразована в отображение Scala с помощью стандартных преобразователей коллекций Java — Scala, которые будут рассмотрены в разделе 24.17.

Изменяемые наборы битов

Изменяемый набор битов похож на неизменяемый. Отличием является то, что он может быть изменен на месте. Изменяемый набор битов по сравнению с неизменяемым работает при обновлениях немного эффективнее, поскольку измененные Long-значения копировать не нужно. Пример использования:

```
scala> val bits = scala.collection.mutable.BitSet.empty
bits: scala.collection.mutable.BitSet = BitSet()

scala> bits += 1
res64: bits.type = BitSet(1)

scala> bits += 3
res65: bits.type = BitSet(1, 3)

scala> bits
res66: scala.collection.mutable.BitSet = BitSet(1, 3)
```

24.10. Массивы

Массивы в Scala являются особой разновидностью коллекции. С одной стороны, массивы Scala в точности соответствуют массивам Java. То есть Scala-массив `Array[Int]` представлен как Java-массив `int[]`, `Array[Double]` — как `double[]`, а `Array[String]` — как `String[]`. Но вместе с тем массивы Scala имеют существенно больше свойств по сравнению с Java-аналогами. Во-первых, массивы Scala могут быть обобщенными. То есть можно воспользоваться массивом `Array[T]`, где `T` является параметром типа или абстрактным типом. Во-вторых, массивы Scala совместимы со Scala-последовательностями, то есть туда, где требуется `Seq[T]`, можно передавать `Array[T]`. И наконец, массивы Scala также поддерживают все операции с последовательностями. Приведем несколько практических примеров:

```
scala> val a1 = Array(1, 2, 3)
a1: Array[Int] = Array(1, 2, 3)

scala> val a2 = a1 map (_ * 3)
a2: Array[Int] = Array(3, 6, 9)

scala> val a3 = a2 filter (_ % 2 != 0)
a3: Array[Int] = Array(3, 9)

scala> a3.reverse
res1: Array[Int] = Array(9, 3)
```

А как с учетом того, что массивы Scala представлены точно так же, как массивы Java, эти дополнительные свойства могут поддерживаться в Scala?

Ответ заключается в систематическом использовании подразумеваемых преобразований. Массив не может претендовать на то, чтобы быть последовательностью, поскольку тип данных, представляющих настоящий массив, не является подтипом типа

Seq. Вместо этого там, где массив будет использоваться в качестве последовательности Seq, он подразумеваемо будет заключен в подкласс класса Seq. Для этого подкласса используется имя scala.collection.mutable.WrappedArray. Вот как это работает:

```
scala> val seq: Seq[Int] = a1
seq: Seq[Int] = WrappedArray(1, 2, 3)

scala> val a4: Array[Int] = seq.toArray
a4: Array[Int] = Array(1, 2, 3)

scala> a1 eq a4
res2: Boolean = true
```

Сеанс работы с интерпретатором показывает, что массивы совместимы с последовательностями благодаря наличию подразумеваемого преобразования из Array в WrappedArray. Для преобразования в обратном направлении, от WrappedArray к Array, можно воспользоваться методом toArray, который определен в классе Traversable. В последней строке приведенного ранее диалога с интерпретатором показано, что заключенный в оболочку массив при его изъятии оттуда с помощью метода toArray возвращает тот же массив, с которого начиналась работа.

Есть еще одно подразумеваемое преобразование, применимое к массивам. Это преобразование просто добавляет к массивам все методы, применимые к последовательностям, но сами массивы в последовательности не превращает. Добавление означает, что массив заключается в другой объект типа ArrayOps, который поддерживает все методы работы с последовательностями. Обычно этот ArrayOps-объект существует весьма непродолжительное время — он становится недоступен после вызова метода для работы с последовательностью, и его место хранения может быть

использовано повторно. Современные виртуальные машины зачастую вообще избегают создания таких объектов.

Разница между этими двумя подразумеваемыми преобразованиями массивов показана в следующем примере:

```
scala> val seq: Seq[Int] = a1
seq: Seq[Int] = WrappedArray(1, 2, 3)

scala> seq.reverse
res2: Seq[Int] = WrappedArray(3, 2, 1)

scala> val ops: collection.mutable.ArrayOps[Int] =
a1
ops:    scala.collection.mutable.ArrayOps[Int]    =
[I(1, 2, 3)

scala> ops.reverse
res3: Array[Int] = Array(3, 2, 1)
```

Как видите, при вызове метода `reverse` в отношении объекта `seq`, относящегося к типу `WrappedArray`, опять будет получен объект типа `WrappedArray`. Это не противоречит здравому смыслу, поскольку массивы, заключенные в оболочку, относятся к типам `Seq`, а вызов метода `reverse` в отношении любого `Seq`-объекта вновь даст `Seq`-объект. В то же время вызов метода `reverse` в отношении значения `ops` класса `ArrayOps` приведет к возвращению значения типа `Array`, а не `Seq`.

Приведенный ранее пример с `ArrayOps` был надуманным, предназначенным лишь для того, чтобы показать разницу с `WrappedArray`. В обычной обстановке вы никогда не стали бы определять значение класса `ArrayOps`, а просто вызвали бы в отношении массива метод из класса `Seq`:

```
scala> a1.reverse
```

```
res4: Array[Int] = Array(3, 2, 1)
```

Объект ArrayOps вставляется автоматически в ходе подразумеваемого преобразования. Следовательно, показанная выше строка кода эквивалентна следующей строке, где метод intArrayOps является преобразованием, которое подразумеваемо вставлялось в предыдущем примере:

```
scala> intArrayOps(a1).reverse
res5: Array[Int] = Array(3, 2, 1)
```

Возникает вопрос, касающийся способа выбора компилятором intArrayOps в показанной выше строке среди других подразумеваемых преобразований в WrappedArray. Ведь оба преобразования отображают массив на тип, поддерживающий метод reverse, указанный во введенном коде. Ответом на данный вопрос являются степени приоритетности этих двух преобразований. У преобразования ArrayOps более высокая степень приоритетности, чем у WrappedArray. Первое преобразование определено в объекте Predef, а второе — в классе scala.LowPriorityImplicits, являющемся родительским классом для Predef. Подразумеваемые действия в подклассах и подобъектах обладают приоритетом над подразумеваемыми действиями в базовых классах. Следовательно, если применимы оба преобразования, будет выбрано то, которое определено в Predef. Очень похожая схема, рассмотренная в разделе 21.7, работает для строк.

Теперь вы знаете, что массивы совместимы с последовательностями и могут поддерживать все операции, применяемые к последовательностям. А как насчет обобщенности? В Java вы не можете воспользоваться записью T[], где T представляет собой параметр типа. Как же тогда представлен имеющийся в Scala тип Array[T]? Фактически такой обобщенный массив, как Array[T], может в ходе выполнения программы стать

любым из восьми имеющихся в Java элементарных типов массивов: `byte[]`, `short[]`, `char[]`, `int[]`, `long[]`, `float[]`, `double[]`, `boolean[]`, — или же стать массивом объектов. Единственным общим типом, охватывающим в ходе выполнения программы все эти типы, является `AnyRef` (или его аналогия `java.lang.Object`), следовательно, это именно тот тип, на который компилятор Scala отображает `Array[T]`. В ходе выполнения программы, когда происходит обращение к элементу массива типа `Array[T]` или обновление этого элемента, производится череда проверок на соответствие типам, благодаря чему определяется действительный тип массива, а затем выполняется корректная операция уже над Java-массивом. Проверки на соответствие типам несколько замедляют операции над массивами. Можно ожидать, что обращения к обобщенным массивам будут в три-четыре раза медленнее обращений к простым массивам или массивам объектов. Следовательно, если требуется максимально высокая производительность, предпочтение следует отдавать конкретным, а не обобщенным массивам.

Но одного представления типа обобщенного массива недостаточно, должен существовать также способ создания обобщенных массивов. А это еще более сложная задача, требующая от вас некоторой помощи. В качестве примера рассмотрим следующую попытку создания метода, работающего с обобщениями и создающего массив:

```
// Это нерабочий код!
def evenElems[T](xs: Vector[T]): Array[T] = {
  val arr = new Array[T]((xs.length + 1) / 2)
  for (i <- 0 until xs.length by 2)
    arr(i / 2) = xs(i)
  arr
}
```

Метод `evenElems` возвращает новый массив, состоящий из тех

элементов используемого в качестве аргумента вектора `xs`, которые находятся в нем на четных позициях. В первой строке тела метода `evenElems` создается получаемый в результате массив, у которого тот же тип элементов, что и у аргумента. Следовательно, в зависимости от фактического параметра типа для `T` это может быть `Array[Int]`, или `Array[Boolean]`, или массив из других элементарных типов, имеющихся в Java, или же массив какого-нибудь ссылочного типа. Но все эти типы имеют в ходе выполнения программы различные представления, поэтому возникает вопрос: как среда выполнения Scala собирается выбирать из них нужное? По сути, сделать это, основываясь на имеющейся информации, она не может, потому что фактический тип, соответствующий параметру типа `T`, на стадии выполнения кода затирается. Так что при попытке откомпилировать показанный ранее код будет получено следующее сообщение об ошибке:

```
error: cannot find class tag for element type T
val arr = new Array[T]((arr.length + 1) / 2)
          ^
```

Здесь вам следует помочь компилятору, подсказав при выполнении кода, какой параметр типа у `evenElems`. Эта подсказка, предназначенная для использования в ходе выполнения кода, принимает форму *признака класса* типа `scala.reflect.ClassTag`. Признак класса дает описание заданного *затираемого типа*, предоставляя конструктору массива исчерпывающую информацию об этом типе.

Во многих случаях компилятор может создавать признак класса самостоятельно. Именно так обстоит дело с конкретным типом вроде `Int` или `String`. Это же распространяется и на определенные обобщенные типы наподобие `List[T]`, где для выстраивания предположения о затираемом типе информации вполне достаточно; в данном примере затираемым типом будет

List.

Для полностью обобщенных вариантов обычно практикуется передача признака класса с помощью контекстного ограничителя, рассмотренного в разделе 21.6. А вот так с помощью контекстного ограничителя может быть исправлено показанное ранее определение:

```
// Это рабочий код
import scala.reflect.ClassTag
def evenElems[T: ClassTag](xs: Vector[T]):
Array[T] = {
  val arr = new Array[T]((xs.length + 1) / 2)
  for (i <- 0 until xs.length by 2)
    arr(i / 2) = xs(i)
  arr
}
```

В этом новом определении компилятор при создании Array[T] ищет признак класса для параметра типа T, то есть он будет искать подразумеваемое значение типа ClassTag[T]. Если оно будет найдено, он воспользуется признаком класса для создания массива нужного вида. В противном случае будет показано сообщение об ошибке, похожее на показанное ранее. Вот как выглядит диалог с интерпретатором, в котором используется метод evenElems:

```
scala> evenElems(Vector(1, 2, 3, 4, 5))
res6: Array[Int] = Array(1, 3, 5)

scala> evenElems(Vector("this", "is", "a", "test",
"run"))
res7: Array[java.lang.String] = Array(this, a,
run)
```

В обоих случаях компилятор Scala автоматически создает

признак класса для типа элемента (сначала `Int`, потом `String`) и передает его подразумеваемому параметру метода `evenElems`. Компилятор может сделать то же самое для всех конкретных типов, но не способен на это, если сам аргумент является еще одним параметром типа без признака класса. Например, следующий код не пройдет компиляцию:

```
scala> def wrap[U](xs: Vector[U]) = evenElems(xs)
<console>:9: error: No ClassTag available for U
     def wrap[U](xs: Vector[U]) = evenElems(xs)
                                            ^
```

Здесь метод `evenElems` получает признак класса для параметра типа `U`, но ничего не находит. Разумеется, решением в данном случае может послужить востребование для `U` еще одного подразумеваемого признака класса. Следующий код уже пройдет компиляцию:

```
scala> def wrap[U: ClassTag](xs: Vector[U]) =
evenElems(xs)
wrap: [U](xs: Vector[U])(implicit evidence$1:
     scala.reflect.ClassTag[U])Array[U]
```

Этот пример показывает также, что контекстное ограничение в определении `U` является краткой формой подразумеваемого параметра, названного здесь `evidence$1` и имеющего тип `ClassTag[U]`.

24.11. Строки

Как и массивы, строки не являются последовательностями в прямом смысле слова, но они могут быть в них преобразованы и также поддерживают все операции с последовательностями. Далее приводятся примеры операций, которые могут вызываться в

отношении строк:

```
scala> val str = "hello"
str: java.lang.String = hello

scala> str.reverse
res6: String = olleh

scala> str.map(_.toUpper)
res7: String = HELLO

scala> str drop 3
res8: String = lo

scala> str slice (1, 4)
res9: String = ell

scala> val s: Seq[Char] = str
s: Seq[Char] = WrappedString(h, e, l, l, o)
```

Эти операции поддерживаются двумя подразумеваемыми преобразованиями, рассмотренными в разделе 21.7. Первое преобразование, имеющее более низкую степень приоритетности, отображает класс `String` на класс `WrappedString`, являющийся подклассом `immutable.IndexedSeq`. Это преобразование было применено в последней строке предыдущего примера, где строка была преобразована в значение типа `Seq`. Другое преобразование с более высокой степенью приоритетности отображает строку на объект `StringOps`, который добавляет к строкам все методы, применяемые к неизменяемым последовательностям. В предыдущем примере это преобразование было подразумеваемо вставлено в вызовы методов `reverse`, `map`, `drop` и `slice`.

24.12. Характеристики производительности

Как показали все предыдущие разъяснения, различные типы коллекций обладают разными характеристиками производительности. Именно это обстоятельство становится главной причиной выбора конкретного типа коллекции из множества других типов. Характеристики производительности некоторых наиболее востребованных операций над коллекциями сведены в табл. 24.10 и 24.11.

Таблица 24.10. Характеристики производительности типов последовательностей

Операции	head	tail	apply	update	prepend	append	insert
Неизменяемые:							
List	C	C	L	L	C	L	–
Stream	C	C	L	L	C	L	–
Vector	eC	eC	eC	eC	eC	eC	–
Stack	C	C	L	L	C	L	–
Queue	aC	aC	L	L	L	C	–
Range	C	C	C	–	–	–	–
String	C	L	C	L	L	L	–
Изменяемые:							
ArrayBuffer	C	L	C	C	aC	L	L
ListBuffer	C	L	L	L	C	C	L
StringBuilder	C	L	C	C	L	aC	L
MutableList	C	L	L	L	C	C	L
Queue	C	L	L	L	C	C	L
ArraySeq	C	L	C	C	–	–	–
Stack	C	L	L	L	C	L	L
ArrayStack	C	L	C	C	aC	L	L
Array	C	L	C	C	–	–	–

Таблица 24.11. Характеристики производительности типов наборов и отображений

Операции	lookup	add	remove	min
Неизменяемые:				
HashSet/HashMap	eC	eC	eC	L
TreeSet/TreeMap	Log	Log	Log	Log
BitSet	C	L	L	eC*
ListMap	L	L	L	L
Изменяемые:				
HashSet/HashMap	eC	eC	eC	L
WeakHashMap	eC	eC	eC	L
BitSet	C	aC	C	eC*

* Если биты плотно упакованы.

Записи в этих двух таблицах расшифровываются следующим образом:

- C — операция занимает постоянное (короткое) время;
- eC — операция занимает практически постоянное время, но это может зависеть от некоторых допущений, таких как максимальная длина вектора или распределение хеш-ключей;
- aC — операция занимает амортизированное постоянное время. Некоторые вызовы операции могут занимать больше времени, но если выполняется множество операций, то берется только постоянное среднее время, затрачиваемое на одну операцию;
- Log — на операцию уходит время, пропорциональное логарифму размера коллекции;
- L — операция имеет линейный характер, то есть на нее уходит время, пропорциональное размеру коллекции;

• – — операция не поддерживается.

В табл. 24.10 рассматриваются как неизменяемые, так и изменяемые типы последовательностей со следующими операциями:

• `head` — выбор первого элемента последовательности;

• `tail` — создание новой последовательности, содержащей все элементы, за исключением первого;

• `apply` — индексирование;

• `prepend` — добавление элемента к началу последовательности. Если последовательность неизменяемая, эта операция приводит к созданию новой последовательности. Если последовательность изменяемая, то существующая последовательность изменяется;

• `append` — добавление элемента к концу последовательности. Если последовательность неизменяемая, эта операция приводит к созданию новой последовательности. Если последовательность изменяемая, то существующая последовательность изменяется;

• `insert` — вставка элемента в произвольную позицию последовательности. Поддерживается только для изменяемых последовательностей.

В табл. 24.11 рассматриваются как неизменяемые, так и изменяемые типы наборов и отображений со следующими операциями:

• `lookup` — проверка на наличие элемента в наборе или выбор значения, связанного с ключом;

• `add` — добавление нового элемента к набору или новой пары

«ключ —значение» к отображению;

- `remove` — удаление всех элементов из набора или ключа из отображения;

- `min` — возвращение наименьшего элемента набора или наименьшего ключа отображения.

24.13. Равенство

В библиотеках коллекций выдерживается единый подход к равенству и хешированию. В первую очередь замысел заключается в разбиении коллекций на наборы, отображения и последовательности. Коллекции из различных категорий всегда не равны. Например, коллекция `Set(1, 2, 3)` не равна коллекции `List(1, 2, 3)` даже при том, что они содержат одни и те же элементы. В то же время внутри одной и той же категории коллекции равны лишь при условии, что у них имеются одинаковые элементы (для последовательностей — одинаковые элементы, стоящие в одинаковом порядке), например `List(1, 2, 3) == Vector(1, 2, 3)` и `HashSet(1, 2) == TreeSet(2, 1)`.

При проверке равенства неважно, является коллекция изменяемой или неизменяемой. Для изменяемых коллекций равенство просто зависит от содержащихся в ней элементов на момент выполнения проверки на равенство. Это означает, что в разное время изменяемая коллекция может быть равна разным коллекциям в зависимости от добавления или удаления элементов. Это обстоятельство может стать ловушкой при использовании изменяемых коллекций в качестве ключа в хеш-отображении. Например:

```
scala> import collection.mutable.{HashMap,
ArrayBuffer}
```

```
import collection.mutable.{HashMap, ArrayBuffer}

scala> val buf = ArrayBuffer(1, 2, 3)
buf: scala.collection.mutable.ArrayBuffer[Int] =
ArrayBuffer(1, 2, 3)

scala> val map = HashMap(buf -> 3)
map:
scala.collection.mutable.HashMap[scala.collection.
mutable.ArrayBuffer[Int],Int]                    =
Map((ArrayBuffer(1, 2, 3),3))

scala> map(buf)
res13: Int = 3

scala> buf(0) += 1

scala> map(buf)
java.util.NoSuchElementException: key not found:
  ArrayBuffer(2, 2, 3)
```

В данном примере выбор, сделанный в последней строке, скорее всего, завершится сбоем, поскольку хеш-код массива `xs` в предпоследней строке изменился. Поэтому при поиске на основе хеш-кода будет отыскиваться другое место, отличное от того, где был сохранен `xs`.

24.14. Отображения

В коллекциях имеется всего несколько методов, создающих новые коллекции. В качестве примеров можно привести `map`, `filter` и `++`. Такие методы мы называем *преобразователями*, поскольку они получают как минимум одну коллекцию в качестве объекта-

получателя и создают в результате своей работы еще одну коллекцию.

Преобразователи могут быть реализованы двумя основными способами — строгим и нестрогим (или «ленивым»). Строгий преобразователь создает новую коллекцию со всеми ее элементами. А нестрогий — только заместителя получаемой в результате коллекции, а ее элементы — по требованию.

В качестве примера нестрогого преобразователя рассмотрим следующую реализацию ленивой операции `map`:

```
def lazyMap[T, U](coll: Iterable[T], f: T => U) =
  new Iterable[U] {
    def iterator = coll.iterator map f }
```

Обратите внимание на то, что `lazyMap` создает новый объект типа `Iterable` без обхода всех элементов заданной коллекции `coll`. Вместо этого заданная функция `f` применяется к элементам итератора `iterator` новой коллекции по мере их востребованности.

По умолчанию коллекции в Scala являются строгими во всех своих проявлениях, за исключением `Stream`, все методы преобразования которой имеют «ленивый» характер. Но существует систематический подход для превращения каждой коллекции в ленивую и наоборот, основанный на представлениях коллекций. Представление является специализированным эквивалентом какой-либо основной коллекции, где все преобразователи реализованы в ленивом варианте.

Для перехода от коллекции к ее представлению можно воспользоваться в отношении коллекции методом `view`. Если `xs` является некой коллекцией, то `xs.view` создает точно такую же коллекцию, но с «ленивой» реализацией всех преобразователей. Чтобы перейти обратно от представления к строгой коллекции, можно воспользоваться методом `force`.

В качестве примера предположим, что имеется вектор Int-значений, в отношении которого нужно одну за другой применить две функции map:

```
scala> val v = Vector(1 to 10: _ * )
v: scala.collection.immutable.Vector[Int] =
  Vector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> v map (_ + 1) map (_ * 2)
res5: scala.collection.immutable.Vector[Int] =
  Vector(4, 6, 8, 10, 12, 14, 16, 18, 20, 22)
```

В последней инструкции выражение v map (_ + 1) создает новый вектор, который вторым вызовом map (_ * 2) преобразуется в третий вектор. Во многих ситуациях создание промежуточного результата из первого вызова map представляется неэкономным. В надуманном примере быстрее было бы воспользоваться одним вызовом map в сочетании с двумя функциями, (_ + 1) и (_ * 2). При наличии двух функций, доступных в одном и том же месте, это можно сделать самостоятельно. Но довольно часто последовательные преобразования структур данных выполняются в различных программных модулях. Объединение этих преобразований может разрушить модульность. Более универсальный способ избавления от промежуточных результатов заключается в том, что сначала вектор превращается в представление, затем к представлению применяются все преобразования, после чего представление опять преобразуется в вектор:

```
scala> (v.view map (_ + 1) map (_ * 2)).force
res12: Seq[Int] = Vector(4, 6, 8, 10, 12, 14, 16,
18, 20, 22)
```

Мы вновь поочередно выполним эту последовательность

операций:

```
scala> val vv = v.view
vv: scala.collection.SeqView[Int,Vector[Int]] =
  SeqView(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Применение `v.view` выдает значение типа `SeqView`, то есть «лениво» вычисляемую `Seq`-последовательность. У типа `SeqView` имеется два параметра типа. Первый, `Int`, показывает тип элементов представления. Второй, `Vector[Int]`, — тип конструктора, получаемого обратно при принудительном переводе в представление.

Применение первого метода `map` к представлению дает следующий результат:

```
scala> vv map (_ + 1)
res13:    scala.collection.SeqView[Int,Seq[_]]    =
SeqViewM(...)
```

Результатом выполнения `map` является значение, которое выглядит на экране как `SeqViewM(...)`. По сути, это оболочка, фиксирующая тот факт, что `map` с функцией `(_ + 1)` нужно применить к вектору `v`. Но этот метод `map` не применяется, пока не будет принудительно создаваться представление. Буква M после `SeqView` показывает, что представление инкапсулирует операцию `map`. Другие буквы обозначают иные отложенные операции, например `S` — отложенную операцию `slice`, а `R` — `reverse`. Теперь применим к последнему результату второй метод `map`:

```
scala> res13 map (_ * 2)
res14:    scala.collection.SeqView[Int,Seq[_]]    =
SeqViewMM(...)
```

Получаем `SeqView` с двумя операциями `map`, поэтому название выводится с двойной буквой M: `SeqViewMM(...)`. И наконец,

принудительное получение последнего результата приводит к следующему диалогу:

```
scala> res14.force
res15: Seq[Int] = Vector(4, 6, 8, 10, 12, 14, 16,
18, 20, 22)
```

Обе сохраненные функции используются в качестве составных частей выполнения операции `force`, и создается новый вектор. При таком способе промежуточные структуры данных не нужны.

Следует обратить внимание еще на одну деталь: статическим типом конечного результата является `Seq`, а не `Vector`. Обратная трассировка типов показывает, что, как только была применена первая отложенная операция `map`, результат получил статический тип `SeqViewM[Int, Seq[_]]`. То есть знание о том, что представление было применено к специализированному типу последовательности `Vector`, оказалось утрачено. Реализация представления для любого конкретного класса требует дополнительного кода, поэтому в библиотеке коллекций Scala имеются представления главным образом для обобщенных типов коллекций, но не для конкретных реализаций126. Потребность в использовании представлений может обусловливаться двумя причинами. В первую очередь это производительность. Вы видели, что переключением коллекции на представление удалось избежать создания промежуточных результатов. Такая экономия может сыграть весьма важную роль. В качестве еще одного примера рассмотрим задачу нахождения первого палиндрома в списке слов. Палиндромом называется слово, которое читается в обратном порядке точно так же, как и в прямом. Необходимые для этого определения имеют следующий вид:

```
def isPalindrome(x: String) = x == x.reverse
def findPalindrome(s: Seq[String]) = s find
isPalindrome
```

Теперь предположим, что имеется весьма длинная последовательность слов и нужно найти палиндром в первом миллионе слов этой последовательности. Можно ли повторно воспользоваться определением `findPalindrome`? Разумеется, можно создать следующий код:

```
findPalindrome(words take 10 00000)
```

Он неплохо разделяет два аспекта, заключающихся в получении первого миллиона слов последовательности и нахождении в них палиндрома. Но недостаток этого решения состоит в неизменном создании промежуточной последовательности, состоящей из миллиона слов, даже если первое слово этой последовательности уже является палиндромом. Следовательно, потенциально получается, что 999 999 слов копируются в промежуточный результат без последующей проверки.

Многие программисты откажутся от этого и напишут собственную специализированную версию нахождения палиндрома в некотором заданном префиксе последовательности аргументов. Но с представлениями этого делать не придется. Нужно просто воспользоваться следующим кодом:

```
findPalindrome(words.view take 10 00000)
```

Здесь также неплохо разделены интересы, но вместо последовательности из миллиона элементов будет создан отдельный легковесный объект представления. Таким образом, вам не придется выбирать между производительностью и модульностью.

Второй практический прием касается применения представлений к изменяемым последовательностям. Многие функции преобразований, применяемые в отношении таких представлений, создают окно в исходную последовательность, которое затем можно использовать для избирательного обновления некоторых элементов этой последовательности. Чтобы

увидеть это на примере, предположим, что имеется некий массив arr:

```
scala> val arr = (0 to 9).toArray
arr: Array[Int] = Array(0, 1, 2, 3, 4, 5, 6, 7, 8,
9)
```

Можно создать подокно в этот массив путем вычленения фрагмента представления этого массива:

```
scala> val subarr = arr.view.slice(3, 6)
subarr: scala.collection.mutable.IndexedSeqView[
  Int,Array[Int]] = IndexedSeqViewS(...)
```

Таким образом будет получено представление subarr, ссылающееся на элементы массива arr с третьей по пятую позиции. Представление не копирует эти элементы, оно только предоставляет ссылку на них. Теперь предположим, что у вас имеется метод, изменяющий некоторые элементы последовательности. Например, следующий метод изменения знака negate проведет эту операцию над всеми элементами указанной последовательности целых чисел:

```
scala> def negate(xs: collection.mutable.Seq[Int])
=
            for (i <- 0 until xs.length) xs(i) = -
xs(i)
negate:                                          (xs:
scala.collection.mutable.Seq[Int])Unit
```

А теперь предположим, что нужно изменить знак в позициях массива arr с третьей по пятую. Можно ли в таком случае применить метод negate? С использованием представления все делается просто:

```
scala> negate(subarr)
```

```
scala> arr
res4: Array[Int] = Array(0, 1, 2, -3, -4, -5, 6,
7, 8, 9)
```

Здесь `negate` вносит изменения во все элементы `subarr`, то есть во фрагмент элементов массива `arr`. И опять же вы видите, что представления помогают сохранять модульность. Показанный ранее код отделяет вопрос о том, к какому диапазону индексов применить метод, от вопроса, какой именно метод применить.

После просмотра всех этих весьма интересных использований представлений может возникнуть вопрос: а зачем вообще нужны строгие коллекции? Одна из причин в том, что сравнение производительности не всегда бывает в пользу «ленивых» коллекций. Для коллекций меньших размеров добавленные издержки на формирование и применение замыканий в представлениях зачастую выше, чем выгоды, получаемые за счет того, что в них не применяются промежуточные структуры данных. Возможно, еще более существенной причиной является то, что вычисление в представлениях могут создавать серьезные помехи, если у отложенных операций имеются побочные эффекты.

Вот пример, которым можно уязвить пользователей Scala до версии 2.8. В этих версиях тип `Range` был «ленивым», поэтому он вел себя наподобие представления. Программисты пробовали создавать такие вот акторы[127]:

```
val actors = for (i <- 1 to 10) yield actor { ...
}
```

А потом удивлялись, почему впоследствии ни один из акторов не выполнялся, даже если метод `actor` должен создавать и запускать актор из следующего за ним кода, заключенного в фигурные скобки. Чтобы понять, почему ничего не происходит, вспомним, что показанное ранее выражение `for` эквивалентно

применению следующего метода `map`:

```
val actors = (1 to 10) map (i => actor { ... })
```

Поскольку ранее диапазон, созданный выражением `(1 to 10)`, вел себя подобно представлению, результатом выполнения `map` опять было представление. То есть элементы не вычислялись, а следовательно, акторы не создавались! Акторы создавались бы принудительным вычислением диапазона всего значения, но не обязательно именно это требовалось для того, чтобы заставить акторов выполнить их работу.

Чтобы избегать подобных сюрпризов, коллекции Scala в версии 2.8 получили более простые правила. Все коллекции, за исключением потоков и представлений, являются строгими. Перейти от строгой коллекции к «ленивой» можно только через метод представления `view`. Единственным способом обратного переходя является применение метода `force`. Следовательно, акторы, определенные ранее, в Scala 2.8 поведут себя так, как от них ожидают, то есть будут созданы и запущены 10 акторов. Чтобы вернуться к прежнему парадоксальному поведению, придется добавить явно выраженный вызов метода `view`:

```
val actors = for (i <- (1 to 10).view) yield actor
{ ... }
```

В целом представления являются весьма мощным инструментом, позволяющим увязать соображения эффективности с соображениями модульности. Но чтобы не путаться в тонкостях отложенного вычисления, применение представлений лучше ограничить двумя сценариями: применять их либо в чисто функциональном коде, где у преобразований коллекций нет побочных эффектов, либо к изменяемым коллекциям, когда все изменения выполняются в явном виде. И лучше избегать смешивания представлений и операций, создающих новые коллекции, если у них к тому же есть побочные эффекты.

24.15. Итераторы

Итератор является не коллекцией, а способом поочередного обращения к элементам коллекции. Двумя основными операциями итератора являются `next` и `hasNext`. В результате вызова `it.next()` будут возвращены следующий элемент итератора и заранее определенное состояние итератора. Затем при повторном вызове `next` в отношении того же итератора будет выдан элемент, следующий за тем, который был возвращен ранее. Если возвращать станет нечего, вызов `next` приведет к выдаче исключения `NoSuchElementException`. Определить исчерпание элементов можно с помощью имеющегося в классе `Iterator` метода `hasNext`.

Наиболее простым способом последовательного перебора всех элементов, возвращаемых итератором, является использование цикла `while`:

```
while (it.hasNext)
  println(it.next())
```

Итераторы в Scala также предоставляют аналоги большинства методов, имеющихся в трейтах `Traversable`, `Iterable` и `Seq`. Например, они поддерживают метод `foreach`, выполняющий заданную процедуру в отношении каждого элемента, возвращенного итератором. При использовании `foreach` показанный ранее цикл можно сократить до следующего кода:

```
it foreach println
```

Как и всегда, в качестве альтернативного синтаксиса для выражений, использующих `foreach`, `map`, `filter` и `flatMap`, можно воспользоваться выражением `for`. То есть еще одним способом показать на устройстве вывода все элементы, возвращенные итератором, мог бы быть следующий код:

```
for (elem <- it) println(elem)
```

Между методом `foreach`, применяемым в отношении итераторов, и таким же методом, применяемым к коллекциям, допускающим обход элементов, есть существенная разница: при вызове в отношении итератора `foreach` оставит итератор в состоянии завершения его работы. Поэтому еще один вызов `next` в отношении того же самого итератора закончится неудачно и приведет к выдаче исключения `NoSuchElementException`. В отличие от этого, при вызове в отношении коллекции `foreach` оставляет номера элементов в коллекции без изменений, если только переданная функция не добавляет или не удаляет элементы, чего делать не рекомендуется, поскольку это легко может привести к неожиданным результатам.

Другие операции, имеющиеся в `Iterator` наряду с `Traversable`, обладают таким же свойством оставлять итератор в состоянии завершения работы. Например, итераторы предоставляют метод `map`, возвращающий новый итератор:

```
scala> val it = Iterator("a", "number", "of", "words")
it: Iterator[java.lang.String] = non-empty iterator

scala> it.map(_.length)
res1: Iterator[Int] = non-empty iterator

scala> res1 foreach println
1
6
2
5

scala> it.next()
```

```
java.util.NoSuchElementException: next on empty iterator
```

Как видите, после вызова map итератор it добрался до конца.

Еще один пример — метод dropWhile, который может использоваться для поиска первого элемента итератора, обладающего определенным свойством. Например, для поиска в ранее показанном итераторе первого слова, имеющего как минимум два символа, можно воспользоваться следующим кодом:

```
scala> val it = Iterator("a", "number", "of", "words")
it: Iterator[java.lang.String] = non-empty iterator

scala> it dropWhile (_.length < 2)
res4: Iterator[java.lang.String] = non-empty iterator

scala> it.next()
res5: java.lang.String = number
```

Еще раз обратите внимание на то, что it был изменен вызовом dropWhile: теперь it указывает на второе слово number в списке. Фактически it и результат res, полученный при выполнении dropWhile, вернут одинаковую последовательность элементов.

Есть только одна стандартная операция, duplicate, позволяющая многократно использовать один и тот же итератор:

```
val (it1, it2) = it.duplicate
```

Вызов метода duplicate возвращает два итератора, каждый из которых возвращает точно такие же элементы, что и итератор it. Эти два итератора работают независимо друг от друга,

перемещение одного из них совершенно не влияет на состояние другого. В отличие от этого, исходный итератор `it` перемещается при вызове `duplicate` в самый конец и становится непригодным для дальнейшего использования.

В целом, итераторы ведут себя подобно коллекциям, если только никогда не обращаться к итератору еще раз после вызова в отношении него какого-либо метода. Библиотеки коллекций Scala выявляют эту особенность с помощью абстракции под названием `TraversableOnce`, которая является обобщающим родительским трейтом для `Traversable` и `Iterator`. Судя по названию, объекты типа `TraversableOnce` допускают обход своих элементов с использованием `foreach`, но состояние таких объектов после обхода не указывается. Если объект типа `TraversableOnce` фактически относится к типу `Iterator`, после обхода он будет указывать на конец коллекции, но если он относится к типу `Traversable`, то не изменит своего состояния. Чаще всего `TraversableOnce` используется в качестве типа аргумента для методов, который могут получать в качестве аргумента итератор либо объект с возможностью обхода. Примером может послужить метод добавления `++` в трейте `Traversable`. Он получает параметр типа `TraversableOnce`, позволяя добавлять элементы, получаемые как из итератора, так и из коллекций, с возможностью обхода.

Все операции над итераторами сведены в табл. 24.12.

Таблица 24.12. Операции в трейте Iterator

Что собой представляет	Чем занимается
Абстрактные методы: it.next()	Возвращает следующий элемент в итераторе и проходит дальше.
it.hasNext	Возвращает true, если может вернуть еще один элемент
Вариации:	Создает накапливающий в буфере итератор, возвращающий все

it.buffered	элементы it
it grouped size	Создает итератор, выдающий элементы, возвращаемые it, в последовательности блоков фиксированного размера.
it sliding size	Создает итератор, выдающий элементы, возвращаемые it, в последовательности, представляемые скользящим по коллекции окном фиксированного размера
Копирование: it copyToBuffer buf	Копирует все элементы, возвращаемые it, в буфер buf.
it copyToArray(arr, s, l)	Копирует не более одного элемента, возвращенного it, в массив arr, начиная с индекса s. Последние два аргумента являются необязательными
Дублирование: it.duplicate	Создает пару итераторов, каждый из которых независимо от другого возвращает все элементы it.
Добавления: it ++ jt	Создает итератор, выдающий все элементы, возвращаемые итератором it, а затем все элементы, возвращаемые итератором jt.
it padTo (len, x)	Создает итератор, выдающий все элементы it, а затем копии x до тех пор, пока не будет полностью достигнута длина len
Отображения: it map f	Возвращает итератор, получаемый в результате применения функции к каждому элементу, возвращаемому it.
it flatMap f	Возвращает итератор, получаемый в результате применения относящейся ко всему итератору функции f к каждому элементу it и дополнения результатов.
it collect f	Возвращает итератор, получаемый в результате использования частично применяемой функции f к каждому элементу it, для которого она определена, и сбора результатов
Преобразования: it.toArray	Собирает возвращаемые it элементы в массив.
it.toList	Собирает возвращаемые it элементы в список.
it.toIterable	Собирает возвращаемые it элементы в коллекцию, допускающую обход элементов.
it.toSeq	Собирает возвращаемые it элементы в последовательность.
it.toIndexedSeq	Собирает возвращаемые it элементы в индексированную последовательность.
it.toStream	Собирает возвращаемые it элементы в поток.
it.toSet	Собирает возвращаемые it элементы в набор.
it.toMap	Собирает возвращаемые it пары «ключ — значение» в отображение

Что собой представляет	Чем занимается
Информация о размере: it.isEmpty	Проверяет, не пустой ли итератор (противоположность hasNext).
it.nonEmpty	Проверяет, содержит ли итератор элементы (псевдоним метода hasNext).
it.size	Выдает количество элементов, возвращенных it (после этой операции it окажется в самом конце!).
it.length	То же самое, что и it.size.
it.hasDefiniteSize	Возвращает true, если известно, что it возвратит конечное число элементов (по определению то же самое, что и isEmpty)
Индексный поиск с извлечением элемента: it find p	Возвращает Option-значение, содержащее первый возвращаемый it-элемент, удовлетворяющий условию p, или None при отсутствии таких элементов (итератор проходит дальше найденного элемента или, если не найден ни один элемент, оказывается в конечной позиции).
it indexOf x	Возвращает индекс первого возвращенного it-элемента, равного x (итератор проходит дальше найденного элемента).
it indexWhere p	Возвращает индекс первого возвращенного it-элемента, удовлетворяющего условию p (итератор проходит дальше этого элемента)
Подитераторы: it take n	Возвращает итератор, выдающий первые *n* элементов итератора it (it проходит дальше позиции *n*-го элемента или же оказывается в конечной позиции, если в it содержится меньше *n* элементов).
it drop n	Возвращает итератор, начинающий выборку с элемента it с позиции (*n* + 1) (it дойдет до этой же позиции).
it slice (m, n)	Возвращает итератор, выдающий блок элементов, возвращенный из it, начинающийся с *m*-го элемента и заканчивающийся перед *n*-м элементом.
it takeWhile p	Возвращает итератор, выдающий элементы из it, пока условие p вычисляется в true.
it dropWhile p	Возвращает итератор, пропускающий элементы из it, пока условие p вычисляется в true, и выдающий остаток.
it filter p	Возвращает итератор, выдающий все элементы из it, удовлетворяющие условию p.
it withFilter p	То же самое, что и it filter p. Эта операция необходима для использования итераторов в выражениях for.

it filterNot p	Возвращает итератор, выдающий все элементы из it, не удовлетворяющие условию p
Подразделы: it partition p	Разбивает it на два итератора, один из которых возвращает из it все элементы, удовлетворяющие условию p, а другой — все элементы, не удовлетворяющие этому условию
Продолжение^	

Таблица 24.12 (продолжение)

Что собой представляет	Чем занимается
Состояние элементов: it forall p	Возвращает булево значение, показывающее, соблюдается ли условие p для всех элементов, возвращаемых it.
it exists p	Возвращает булево значение, показывающее, соблюдается ли условие p для какого-либо из элементов, возвращаемых it.
it count p	Возвращает количество элементов в it, удовлетворяющих условию p
Свертки: (z /: it)(op)	Применяет бинарную операцию op между смежными элементами, возвращаемыми it, проходя коллекцию слева направо и начиная с позиции z.
(it :\ z)(op)	Применяет бинарную операцию op между смежными элементами, возвращаемыми it, проходя коллекцию справа налево и начиная с позиции z.
it.foldLeft(z)(op)	То же самое, что и (z /: it)(op).
it.foldRight(z)(op)	То же самое, что и (it :\ z)(op).
it reduceLeft op	Применяет бинарную операцию op между смежными элементами, возвращаемыми непустым итератором it, проходя коллекцию слева направо.
it reduceRight op	Применяет бинарную операцию op между смежными элементами, возвращаемыми непустым итератором it, проходя коллекцию справа налево
Специализированные свертки: it.sum	Возвращает сумму значений числовых элементов, выдаваемых итератором it.
it.product	Возвращает произведение значений числовых элементов, выдава-емых итератором it.
	Возвращает минимальное значение упорядоченных элементов,

it.min	выдаваемых итератором it.
it.max	Возвращает максимальное значение упорядоченных элементов, выдаваемых итератором it
Объединители: it zip jt	Возвращает итератор пары соответствующих элементов, выдаваемых итераторами it и jt.
it zipAll (jt, x, y)	Возвращает итератор пары соответствующих элементов, выдаваемых итераторами it и jt, причем тот итератор, что короче, расширяется, чтобы соответствовать более длинному итератору путем добавления элементов x или y.
it.zipWithIndex	Возвращает итератор пар, состоящих из элементов, возвращаемых it, и их индексов
Обновление: it patch (i, jt, r)	Возвращает итератор, получаемый из it путем замены r элементов, начиная с позиции i, итератором вставки jt
Что собой представляет	**Чем занимается**
Сравнение: it sameElements jt	Проверяет, не возвращают ли итераторы it и jt одни и те же элементы в одном и том же порядке (как минимум один из итераторов it и jt после этой операции окажется в своей конечной позиции)
Строки: it addString (b, start, sep, end)	Добавляет строку к StringBuilder b, которая показывает все элементы, возвращаемые it, между разделителями sep и заключена в строковые значения start и end. Аргументы start, sep и end являются необязательными
it mkString (start, sep, end)	Преобразует итератор в строку, показывающую все элементы, возвращаемые it, между разделителями sep, и заключенную в строковые значения start и end. Аргументы start, sep и end являются необязательными

Буферированные итераторы. Порой бывает необходим итератор, способный заглянуть вперед, позволяя проинспектировать следующий возвращаемый элемент, не проходя его в процессе перемещения. Рассмотрим, к примеру, задачу пропуска лидирующих пустых строк из итератора, возвращающего последовательность строк. Возможно, возникнет соблазн создать нечто похожее на следующий метод:

```
// Этот код работать не будет
```

```
def skipEmptyWordsNOT(it: Iterator[String]) = {
  while (it.next().isEmpty) {}
}
```

Но, присмотревшись к этому коду, можно понять, что он неработоспособен: разумеется, код будет пропускать лидирующие пустые строки, но он также продвинет `it` за первую непустую строку!

Решение этой задачи заключается в использовании буферированного итератора, экземпляра трейта `BufferedIterator`, который является подтрейтом трейта `Iterator` и предоставляет еще один метод по имени `head`. Вызов `head` в отношении буферированного итератора приведет к возвращению его первого элемента, но без продвижения итератора. При использовании буферированного итератора код для пропуска пустых слов может выглядеть следующим образом:

```
def skipEmptyWords(it: BufferedIterator[String]) =
  while (it.head.isEmpty) { it.next() }
```

Каждый итератор может быть преобразован в буферированный итератор вызовом своего метода `buffered`. Пример его использования выглядит следующим образом:

```
scala> val it = Iterator(1, 2, 3, 4)
it: Iterator[Int] = non-empty iterator

scala> val bit = it.buffered
bit: java.lang.Object with scala.collection.
BufferedIterator[Int] = non-empty iterator
scala> bit.head
res10: Int = 1

scala> bit.next()
```

```
res11: Int = 1

scala> bit.next()
res11: Int = 2
```

Следует заметить, что вызов `head` в отношении буферированного итератора `bit` не приводит к изменению позиции итератора. Поэтому последующий вызов, `bit.next()`, возвращает то же самое значение, что и `bit.head`.

24.16. Создание коллекции с нуля

Вам уже попадался синтаксис вроде `List(1, 2, 3)`, создающий список из трех целых чисел, и `Map('A' -> 1, 'C' -> 2)`, который создает отображение с двумя связками. Фактически это универсальная возможность коллекций Scala. Можно взять любое имя коллекции и указать после него в круглых скобках перечень элементов.

В результате получится новая коллекция с заданными элементами. Вот еще ряд примеров:

```
Traversable()                    // Пустой объект с возможностью
                                 // обхода элементов
List()                           // Пустой список
List(1.0, 2.0)                   // Список с элементами 1.0, 2.0
Vector(1.0, 2.0)                 // Вектор с элементами 1.0, 2.0
Iterator(1, 2, 3)                // Итератор, возвращающий три целых числа
Set(dog, cat, bird)              // Набор из трех животных
HashSet(dog, cat, bird)          // Хеш-набор из тех
```

```
же животных
Map('a' -> 7, 'b' -> 0)          // Отображение
символов на целые числа
```

Скрытно каждая из показанных ранее строк является вызовом метода `apply` определенного объекта. Например, третья из этих строк раскрывается в следующий код:

```
List.apply(1.0, 2.0)
```

Здесь показан вызов метода `apply`, принадлежащего объекту-спутнику класса `List`. Этот метод получает произвольное число аргументов и создает из них список. Каждый класс коллекций в библиотеке Scala располагает объектом-спутником с таким же методом `apply`. И неважно, представлена конкретная реализация классом коллекции, как в случае с `List`, `Stream` или `Vector`, или же трейтом, как в случае с `Seq`, `Set` или `Traversable`. В последнем случае вызов `apply` приведет к созданию некой исходной реализации на основе трейта. Рассмотрим ряд примеров:

```
scala> List(1, 2, 3)
res17: List[Int] = List(1, 2, 3)
scala> Traversable(1, 2, 3)
res18: Traversable[Int] = List(1, 2, 3)

scala> mutable.Traversable(1, 2, 3)
res19: scala.collection.mutable.Traversable[Int] =
  ArrayBuffer(1, 2, 3)
```

Кроме `apply` в каждом объекте-спутнике определен также элемент `empty`, возвращающий пустую коллекцию. Поэтому вместо `List()` можно воспользоваться кодом `List.empty`, вместо `Map()` использовать код `Map.empty` и т. д.

Потомки трейта `Seq` в своих объектах-спутниках предоставляют

также другие операции-фабрики, которые сведены в табл. 24.13. Вкратце это:

- `concat`, которая объединяет произвольное количество объектов, допускающих обход элементов;
- `fill` и `tabulate`, которые создают одно- или многоразмерные последовательности заданной размерности, инициализированные каким-либо выражением или функцией составления таблицы;
- `range`, которая создает последовательность целых чисел с какой-либо постоянной длиной шага;
- `iterate`, которая создает последовательность, получающуюся из многократного применения функции к начальному элементу.

Таблица 24.13. Фабричные методы для последовательностей

Что собой представляет	Чем занимается
S.empty	Создает пустую последовательность
S(x, y, z)	Создает последовательность, состоящую из элементов x, y и z
S.concat(xs, ys, zs)	Создает последовательность, получаемую объединением элементов xs, ys и zs
S.fill(n)(e)	Создает последовательность длиной *n*, где каждый элемент вычисляется выражением e
S.fill(m, n)(e)	Создает последовательность последовательностей размерностью *m* × *n*, где каждый элемент вычисляется выражением e (существует также в более высоких измерениях)
S.tabulate(n)(f)	Создает последовательность длиной *n*, где элемент по каждому индексу i вычисляется путем вызова f(i)
S.tabulate(m, n)(f)	Создает последовательность последовательностей размерностью *m* × *n*, где элемент по каждому индексу (i, j) вычисляется путем вызова f(i, j) (существует также в более высоких измерениях)
S.range(start, end)	Создает последовательность целых чисел start … end – 1

S.range(start, end, step)	Создает последовательность целых чисел, начинающуюся со start и наращиваемую с шагом step до значения end, исключая само это значение
S.iterate(x, n)(f)	Создает последовательность длиной *n* с элементами x, f(x), f(f(x))

24.17. Преобразования между коллекциями Java и Scala

Как и в Scala, в Java имеется богатая библиотека коллекций. У этих двух библиотек много общего. Например, и в той и в другой имеются такие категории, как итераторы, итерируемые коллекции, наборы, отображения и последовательности. Но есть и важные различия. В частности, в библиотеках Scala уделяется намного больше внимания неизменяемым коллекциям и предоставляется намного больше операций, выполняющих преобразование коллекции в новую коллекцию.

Иногда может понадобиться выполнить преобразование из одной среды в другую. Например, появится необходимость обратиться к уже существующей Java-коллекции, как если бы это была Scala-коллекция. Или же может потребоваться передать одну из коллекций Scala методу Java, который ожидает получения Java-аналога. Сделать это не составит никакого труда, поскольку Scala предлагает в объекте `JavaConversions` подразумеваемые преобразования между всеми основными типами коллекций. В частности, двунаправленные преобразования имеются между следующими типами:

```
Iterator          <->     java.util.Iterator
Iterator          <->     java.util.Enumeration
Iterable          <->     java.lang.Iterable
Iterable          <->     java.util.Collection
mutable.Buffer    <->     java.util.List
mutable.Set       <->     java.util.Set
mutable.Map       <->     java.util.Map
```

Чтобы включить эти преобразования, нужно просто импортировать их поддержку:

```
scala> import collection.JavaConversions._
import collection.JavaConversions._
```

Теперь будет действовать автоматическое преобразование между коллекциями Scala и соответствующими им коллекциями Java:

```
scala> import collection.mutable._
import collection.mutable._

scala> val jul: java.util.List[Int] = ArrayBuffer(1, 2, 3)
jul: java.util.List[Int] = [1, 2, 3]

scala> val buf: Seq[Int] = jul
buf: scala.collection.mutable.Seq[Int] = ArrayBuffer(1, 2, 3)

scala> val m: java.util.Map[String, Int] =
          HashMap("abc" -> 1, "hello" -> 2)
m: java.util.Map[String,Int] = {hello=2, abc=1}
```

Внутренний механизм этих преобразований работает за счет создания объекта-оболочки, пересылающего все операции базовому объекту коллекции. Поэтому коллекции при преобразовании между Java и Scala никогда не копируются. Интересной особенностью является то, что при круговом преобразовании из, скажем, Java-типа в соответствующий Scala-тип и обратно, в тот же Java-тип, будет получен точно такой же объект коллекции, который имелся в самом начале.

Есть также ряд других востребованных Scala-коллекций,

которые могут быть преобразованы в Java-типы, но для которых нет соответствующих преобразований в обратном направлении. К ним относятся:

```
Seq             ->     java.util.List
mutable.Seq     ->     java.util.List
Set             ->    java.util.Set
Map             ->    java.util.Map
```

Поскольку в Java изменяемые и неизменяемые коллекции по типам не различаются, преобразование из, скажем, `collection.immutable.List` выдаст коллекцию `java.util.List`. При всех попытках применения операций по внесению изменений в отношении этой коллекции будет выдаваться исключение `UnsupportedOperationException`, например:

```
scala> val jul: java.util.List[Int] = List(1, 2, 3)
jul: java.util.List[Int] = [1, 2, 3]

scala> jul.add(7)
java.lang.UnsupportedOperationException
                                                    at java.util.AbstractList.add(AbstractList.java:131)
```

Резюме

Теперь вы получили более детальное представление об использовании коллекций Scala. В этих коллекциях применен подход, предоставляющий вам целый ряд не просто полезных специализированных методов, но по-настоящему эффективных строительных блоков. Сочетание двух или трех таких строительных блоков позволит выполнить огромное количество вычислений.

Эффективность стиля, принятого в библиотеке, наиболее ярко проявляется благодаря тому, что в Scala имеется облегченный синтаксис для функциональных литералов, а также тому, что сам язык предоставляет множество типов коллекций, сохраняющих постоянство и неизменяемость.

В данной главе коллекции были показаны с точки зрения программиста, использующего библиотеку коллекций. В следующей главе мы рассмотрим, как создаются коллекции и как могут добавляться ваши собственные типы коллекций.

122 Описание частично определенных функций дано в разделе 15.7.

123 Хеш-извлечения описываются в разделе 24.8.

124 Термин Trie происходит от слова retrieval — «извлекаемый» и произносится «три» или «трай».

125 Появляющийся в ответах интерпретатора в этом и в нескольких других примерах раздела buf.type является синглтон-типом. В разделе 29.6 будет разъяснено, что buf.type означает, что переменная содержит именно тот объект, на который указывает buf.

126 Исключением из этого правила являются массивы — применение отложенных операций опять даст результат со статическим типом Array.

127 Библиотека акторов Scala устарела, но этот исторический прием все еще актуален.

25. Архитектура коллекций Scala

В этой главе дается подробное описание архитектуры среды коллекций Scala. Продолжая изучать тему, начатую в главе 24, вы узнаете подробности внутренней работы среды. Кроме того, будет показано, как эта архитектура помогает в определении ваших собственных коллекций всего в нескольких строках кода с учетом того, что подавляющая часть функциональных возможностей коллекции будет взята из этой среды.

В главе 24 перечислялось большое количество операций над коллекциями, существующих в единой форме во многих реализациях различных коллекций. Реализация заново каждой операции над коллекцией для каждого типа коллекции привела бы к образованию огромного объема кода, основная часть которого была бы скопирована из другого места. Такое дублирование кода со временем может привести к несовместимости, вызванной тем, что операция добавляется или изменяется в одной части библиотеки коллекций, а остальные части остаются без изменений. Главная цель проекта новой структуры коллекций заключалась в том, чтобы избежать дублирования, определяя каждую операцию в наименьшем количестве мест[128]. Подход к проектированию состоял в реализации большинства операций в шаблонах коллекций, которые могли бы гибко наследоваться отдельными базовыми классами и реализациями. В этой главе будут рассмотрены такие шаблоны наряду с другими классами и трейтами, составляющими строительные блоки среды, а также поддерживаемые ими принципы конструирования.

25.1. Построители

Почти все операции над коллекциями реализуются в понятиях *обходчиков* и *построителей*. Обходчиками занимается имеющийся в трейте `Traversable` метод `foreach`, а построителями новых

коллекций — экземпляры класса `Builder`. Немного сокращенное схематичное представление этого класса показано в листинге 25.1.

Листинг 25.1. Схематичное представление класса Builder

```
package scala.collection.generic
class Builder[-Elem, +To] {
  def +=(elem: Elem): this.type
  def result(): To   def clear()
      def   mapResult[NewTo](f:   To   =>   NewTo):
Builder[Elem, NewTo]
    = ...
}
```

Добавить к построителю `b` элемент `x` можно с помощью выражения `b += x`. Этот же синтаксис используется для добавления сразу нескольких элементов: например, `b += (x, y)` и `b ++= xs` работают, как с буферами. (Фактически буферы являются расширенными версиями построителей.) Метод `result()` возвращает коллекцию из построителя. Состояние построителя после получения его результата становится неопределенным, но оно может быть перезапущено в новое пустое состояние с помощью метода `clear()`. Построители обобщены как по типу элемента, `Elem`, так и по типу возвращаемой ими коллекции, `To`.

Довольно часто построитель для сбора элементов коллекции может ссылаться на какой-нибудь другой построитель, собираясь затем выполнить преобразование результата другого построителя, чтобы, к примеру, присвоить ему другой тип. Решение этой задачи упрощается за счет использования метода `mapResult` из класса `Builder`. Например, предположим, что имеется буфер массива `buf`. Буферы массивов сами для себя являются построителями, поэтому получение результата с помощью метода `result()` буфера массива приведет к возвращению того же самого буфера.

Если нужно воспользоваться этим буфером для создания построителя массивов, можно прибегнуть к методу mapResult:

```
scala> val buf = new ArrayBuffer[Int]
buf: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer()

scala> val bldr = buf mapResult (_.toArray)
bldr: scala.collection.mutable.Builder[Int,Array[Int]]
  = ArrayBuffer()
```

Получившееся значение bldr является построителем, использующим для сбора элементов буфер массива buf. Когда от bldr требуется получение результата, вычисляется результат buf, выдающий сам буфер массива buf. Этот буфер массива затем отображается с помощью _.toArray на массив. Следовательно, в результате bldr становится построителем массивов.

25.2. Вынесение за скобки общих операций

Главными целями перепроектирования библиотеки коллекций было не только получение естественных типов, но и достижение максимально возможного совместного использования кода реализации. В частности, коллекции Scala придерживаются принципа того же типа результата: везде, где только возможно, метод преобразования, применяемый к коллекции, выдает коллекцию того же типа. Например, операция filter применительно к каждому типу коллекции должна выдавать экземпляр того же типа коллекции. Применение filter к коллекции типа List должно давать коллекцию типа List, применение к Map должно давать Map и т. д. Далее в этом разделе будет показано, каким образом достигается соблюдение этого принципа.

Ускоренный режим чтения

Материал данного раздела очень сложен для понимания. Если хочется поскорее освоить тему главы, можно сразу перейти к разделу 25.3 и научиться внедрять в среду свои собственные классы коллекций на конкретных примерах.

Избегать дублирования кода в библиотеке коллекций Scala и придерживаться принципа того же типа результата удается за счет обобщенных построителей и обходчиков, применяемых к коллекциям в так называемых *трейтах реализации*. В именах этих трейтов используется суффикс `Like`: например, `IndexedSeqLike` является трейтом реализации для `IndexedSeq`, аналогично этому `TraversableLike` является трейтом реализации для `Traversable`. Классы коллекций, такие как `Traversable` или `IndexedSeq`, наследуют все свои реализации конкретных методов из этих трейтов. Трейты реализации, в отличие от обычных коллекций, имеют не один, а два параметра типа. Параметризации подвергается не только тип элементов коллекции, но и *тип представления* коллекции (то есть тип исходной коллекции), такой как `Seq[I]` или `List[T]`.

Вот как выглядит, к примеру, заголовок трейта `TraversableLike`:

```
trait TraversableLike[+Elem, +Repr] { ... }
```

Параметр типа `Elem` представляет тип элемента коллекции, допускающей обход, а параметр типа `Repr` обозначает тип его представления. На `Repr` не накладывается никаких ограничений. В частности, из `Repr` может быть получен экземпляр типа, сам по себе не являющийся подтипом типа `Traversable`. Таким образом, классы вне иерархии коллекций, такие как `String` и `Array`, могут все же использовать все операции, определенные в трейте реализации коллекции.

Если взять за пример операцию filter, то она определена единожды для всех классов коллекций в трейте TraversableLike. Выдержка из соответствующего кода показана в листинге 25.2. В трейте объявляются два абстрактных метода, newBuilder и foreach, реализация которых выполняется в конкретных классах коллекций. Операция filter реализована одинаково для всех коллекций, использующих эти методы. Сначала она создает новый построитель для типа представления Repr, используя newBuilder. Затем обходит все элементы текущей коллекции, используя foreach. Если элемент x удовлетворяет заданному условию p, то есть p(x) равно true, он добавляется к коллекции с помощью построителя. И наконец, элементы, собранные в построителе, возвращаются в качестве экземпляра типа коллекции Repr путем вызова имеющегося в построителе метода result.

Листинг 25.2. Реализация метода filter в трейте TraversableLike

```
package scala.collection
trait TraversableLike[+Elem, +Repr] {
  def newBuilder: Builder[Elem, Repr] // deferred
  def foreach[U](f: Elem => U) // deferred
          ...
  def filter(p: Elem => Boolean): Repr = {
    val b = newBuilder
    foreach { elem => if (p(elem)) b += elem }
    b.result
  }
}
```

Применяемая в отношении коллекций операция map реализуется немного сложнее. Например, если f является функцией преобразования String в Int, а xs относится к типу List[String], тогда выражение xs map f выдаст тип List[Int].

Аналогично, если `ys` относится к типу `Array[String]`, тогда выражение `ys map f` должно выдать тип `Array[Int]`. Но как достичь этого без дублирования определения метода `map` в списках и массивах?

Среды `newBuilder`/`foreach`, показанной в листинге 25.2, для этого недостаточно, поскольку она допускает лишь создание новых экземпляров того же самого типа коллекции, а операции `map` необходим экземпляр той же коллекции, что и у конструктора типа, но, возможно, с другим типом элемента. Более того, даже получающийся конструктор типа такой функции, как `map`, при использовании нетривиальных способов может зависеть от других типов аргументов. Рассмотрим пример:

```
scala> import collection.immutable.BitSet
import collection.immutable.BitSet

scala> val bits = BitSet(1, 2, 3)
bits: scala.collection.immutable.BitSet = BitSet(1, 2, 3)

scala> bits map (_ * 2)
res13: scala.collection.immutable.BitSet = BitSet(2, 4, 6)

scala> bits map (_.toFloat)
res14: scala.collection.immutable.Set[Float] =
  Set(1.0, 2.0, 3.0)
```

Если с помощью операции `map` отобразить функцию удвоения `_ * 2` на набор битов, будет получен еще один набор битов. Но, если с применением `map` на тот же набор битов отобразить функцию `(_.toFloat)`, результатом в конечном счете будет тип `Set[Float]`. Разумеется, он не может быть набором битов,

поскольку такой набор содержит Int-, но не Float-значения.

Обратите внимание на то, что тип результата применения операции map зависит от типа переданной ей функции. Если тип результата этой функции, переданной в качестве аргумента, опять будет Int, то результатом применения map будет BitSet. Но если тип результата функции, переданной в качестве аргумента, будет каким-либо другим, результатом применения map станет просто Set. Вскоре вы поймете, как именно в Scala достигается такая гибкость в задании типов.

Проблема с BitSet не является единичным случаем. Вот еще два диалога с интерпретатором, и в обоих функция отображается с применением map на отображение:

```
scala> Map("a" -> 1, "b" -> 2) map { case (x, y)
=> (y, x) }
res3:
scala.collection.immutable.Map[Int,java.lang.String
=
  Map(1 -> a, 2 -> b)

scala> Map("a" -> 1, "b" -> 2) map { case (x, y)
=> y }
res4: scala.collection.immutable.Iterable[Int] =
  List(1, 2)
```

Первая функция меняет местами два аргумента пар «ключ — значение». В результате отображения этой функции опять получается отображение. Теперь пойдем в обратном направлении. Фактически первое выражение выдает инверсию исходного отображения при условии, что оно может быть инвертировано. Вторая функция отображает пару «ключ — значение» на целочисленное значение, а именно на компонент значения данного отображения. В этом случае сформировать тип Map из результатов невозможно, но тип, являющийся родительским

трейтом Map, все же может быть сформирован, и им будет тип Iterable.

Возникает вопрос: почему бы не ограничить map таким образом, чтобы эта операция всегда возвращала тот же самый тип коллекции? Например, в отношении наборов битов map смогла бы принимать только функции Int-на-Int, а в отношении отображений — функции «пары-на-пары». Такие ограничения нежелательны не только с точки зрения объектно-ориентированного моделирования — они противоречат принципу подстановки Лисков: тип Map является подтипом типа Iterable. Следовательно, каждая операция, допустимая для типа Iterable, должна быть допустимой и для типа Map.

В Scala данная проблема решается переопределением, но не в той простой форме переопределения, унаследованной от Java (гибкости которой недостаточно), а более рациональной формой переопределения, реализуемой за счет использования подразумеваемых параметров.

В листинге 25.3 показана реализация map в трейте TraversableLike. Она очень похожа на реализацию filter, представленную в листинге 25.2. Принципиальная разница состоит в том, что там, где в filter использовался метод newBuilder, являющийся в классе TraversableLike абстрактным, в map используется *фабрика построителя* (builder factory, bf), которой передается дополнительный подразумеваемый параметр, относящийся к типу CanBuildFrom.

Листинг 25.3. Реализация map в TraversableLike

```
def map[B, That](f: Elem => B)
      (implicit bf: CanBuildFrom[Repr, B, That]):
That = {
  val b = bf(this)
  for (x <- this) b += f(x)
```

```
  b.result
}
```

В листинге 25.4 показано определение трейта CanBuildFrom, представляющего фабрики построителя. У него три параметра типа: Elem показывает тип элемента, To — тип создаваемой коллекции и From — тип, для которого применяется эта фабрика построителя. Указав правильные подразумеваемые определения фабрик построителя, можно при задании типов приспособить их под нужное поведение.

Листинг 25.4. Трейт CanBuildFrom

```
package scala.collection.generic
trait CanBuildFrom[-From, -Elem, +To] {
  // Создает новый построитель
  def apply(from: From): Builder[Elem, To]
}
```

Возьмем в качестве примера класс BitSet. В его объекте-спутнике должна содержаться фабрика построителя типа CanBuildFrom[BitSet, Int, BitSet]. Это означает, что при работе с BitSet можно создавать другой объект BitSet при условии, что типом создаваемой коллекции является Int. Если это не так, всегда можно вернуться к другой подразумеваемой фабрике построителя, на этот раз реализованной в объекте-спутнике, относящемся к mutable.Set. Типом этой более общей фабрики построителя, где A является обобщенным параметром типа, является

```
CanBuildFrom[Set[_], A, Set[A]]
```

Это означает, что при работе с произвольным набором Set, выраженным подстановочным типом Set[_], можно опять

создавать набор Set независимо от того, каким будет тип элемента, обозначенный как A. Благодаря этим двум подразумеваемым экземплярам CanBuildFrom затем можно будет на основе действующих в Scala правил для подразумеваемого разрешения выбрать тот из них, который лучше подойдет и будет максимально конкретизирован.

Таким образом, подразумеваемое разрешение предоставляет корректные статические типы для сложных операций с коллекциями, таких как map. Ну а как насчет динамических типов? Предположим, что имеется списочное значение, статическим типом которого является отношение к Iterable, и некая функция отображается на это значение:

```
scala> val xs: Iterable[Int] = List(1, 2, 3)
xs: Iterable[Int] = List(1, 2, 3)

scala> val ys = xs map (x => x * x)
ys: Iterable[Int] = List(1, 4, 9)
```

Статическим типом для показанного ранее значения является, как и ожидалось, Iterable. Но его динамическим типом (как и должно быть) по-прежнему является List! Такое поведение достигается еще одним косвенным действием. В качестве аргумента исходной коллекции передается метод apply из CanBuildFrom. Большинство фабрик построителей для обобщенных коллекций с возможностью обхода элементов (фактически все, за исключением фабрик построителей для классов-листьев) перенаправляют вызов к имеющемуся в коллекции методу genericBuilder. Тот, в свою очередь, вызывает построитель, принадлежащий коллекции, в которой он определен. Таким образом Scala использует статическое подразумеваемое разрешение для того, чтобы определиться с ограничениями типов map, и проводит виртуальную диспетчеризацию для выбора наилучшего динамического типа,

соответствующего этим ограничениям.

25.3. Внедрение новых коллекций

Что нужно сделать для внедрения нового класса коллекции, чтобы он воспользовался всеми предопределенными операциями над нужными типами? В этом разделе будут рассмотрены два примера того, как это делается.

Внедрение последовательностей

Предположим, что нужно создать новый тип последовательности для РНК-нитей (RNA strands), представленный последовательностями оснований `A` (аденина), `T` (тимина), `G` (гуанина) и `U` (урацила). Как показано в листинге 25.5, определения для оснований легко настраиваются.

Листинг 25.5. РНК-основания

```
abstract class Base
case object A extends Base
case object T extends Base
case object G extends Base
case object U extends Base

object Base {
  val fromInt: Int => Base = Array(A, T, G, U)
  val toInt: Base => Int = Map(A -> 0, T -> 1, G -
> 2, U -> 3)
}
```

Каждое основание определяется как case-объект, являющийся наследником общего абстрактного класса `Base`. У класса `Base`

имеется объект-спутник, определяющий две функции, создающие отображение между основаниями и целыми числами в диапазоне от 0 до 3. В примере можно увидеть два различных способа использования коллекций для реализации этих функций. Функция `toInt` реализована как `Map`-отображение от `Base`-значений (оснований) на целые числа. Обратная функция, `fromInt`, представлена в виде массива. Здесь используется тот факт, что и отображения, и массивы являются функциями, поскольку наследуются из трейта `Function1`.

Следующей задачей будет определение класса для нитей РНК. Концептуально нить РНК относится к типу `Seq[Base]`. Но нити РНК могут получаться очень длинными, поэтому есть смысл приложить некоторые усилия для их компактного представления. Поскольку оснований только четыре, основание может быть идентифицировано с помощью двух битов, что позволит сохранять в целом числе в виде значений, состоящих из двух битов, 16 оснований. То есть замысел состоит в том, чтобы сконструировать особый подкласс `Seq[Base]`, использующий упакованное представление.

Первая версия этого класса, которая далее будет уточняться, представлена в листинге 25.6. В классе `RNA1` имеется конструктор, получающий в качестве первого аргумента массив `Int`-значений. В этом массиве содержатся упакованные `RNA`-данные, каждый элемент поддерживает 16 оснований, исключение составляет последний элемент массива, который может быть заполнен частично. Второй аргумент, `length`, указывает общее количество оснований в массиве (и в последовательности). Класс `RNA1` расширяет `IndexedSeq[Base]`. В трейте `IndexedSeq`, который берется из пакета `scala.collection.immutable`, определяются два абстрактных метода, `length` и `apply`.

Листинг 25.6. Класс РНК-нитей, первый вариант

```
import collection.IndexedSeqLike
import collection.mutable.{Builder, ArrayBuffer}
import collection.generic.CanBuildFrom

final class RNA1 private (val groups: Array[Int],
    val length: Int) extends IndexedSeq[Base] {

  import RNA1._

  def apply(idx: Int): Base = {
    if (idx < 0 || length <= idx)
      throw new IndexOutOfBoundsException
     Base.fromInt(groups(idx / N) >> (idx % N * S)
& M)
  }
}

object RNA1 {
     //  Количество  разрядов,  необходимое  для
представления группы
  private val S = 2

  // Количество групп, помещающихся в Int-значение
  private val N = 32 / S
  // Битовая маска для изоляции группы
  private val M = (1 << S) - 1

  def fromSeq(buf: Seq[Base]): RNA1 = {
     val groups = new Array[Int]((buf.length + N -
1) / N)
    for (i <- 0 until buf.length)
       groups(i / N) |= Base.toInt(buf(i)) << (i %
```

```
N * S)
    new RNA1(groups, buf.length)
  }
  def apply(bases: Base * ) = fromSeq(bases)
}
```

Они должны быть реализованы в конкретных подклассах. Класс `RNA1` реализует `length` автоматически путем определения параметрического поля (рассмотрено в разделе 10.6) с точно таким же именем. Метод индексации `apply` реализуется с применением кода, показанного в листинге 25.6. По сути, `apply` сначала извлекает целочисленное значение из массива `groups`, затем из этого целого числа извлекает нужное двухбитовое число, используя сдвиг вправо (>>) и маску (&). Закрытые константы `S`, `N` и `M` происходят из объекта-спутника `RNA1`. В `S` указывается размер каждого пакета (то есть два), в `N` — количество пакетов их двух битов, приходящихся на целое число, а `M` представляет собой битовую маску, изолирующую самые младшие `S` разрядов в двоичном слове.

Обратите внимание на то, что конструктор класса `RNA1` является закрытым (private). Это означает, что клиенты не могут создавать `RNA1`-последовательности путем вызова `new`, что вполне резонно, поскольку тем самым от пользователя скрывается представление `RNA1`-последовательностей в виде массивов упакованных элементов. Если клиент не может увидеть, что представляют собой подробности представления последовательности РНК, появляется возможность изменять эти детали представления в любое время в будущем, никак не влияя на клиентский код.

Иными словами, конструкция получает ценную разобщенность интерфейса РНК-последовательностей и их реализации. Но, если создавать РНК-последовательности с помощью `new` невозможно, должен быть какой-то другой способ создания новых РНК-последовательностей или же весь класс станет практически бесполезным. Для создания РНК-последовательностей есть два

альтернативных способа, и оба они предоставляются объектом-спутником RNA1. Первым способом является применение метода fromSeq, который преобразует заданную последовательность оснований (то есть значение типа Seq[Base]) в экземпляр класса RNA1. Метод fromSeq выполняет эту задачу путем упаковки всех оснований, содержащихся в его аргументе в виде последовательности, в массив с последующим вызовом закрытого конструктора, имеющегося в RNA1, с передачей массива и длины исходной последовательности в качестве аргументов. Здесь используется тот факт, что закрытый конструктор класса просматривается из объекта-спутника класса.

Второй способ создания RNA1-значения предоставляется методом apply, имеющимся в объекте RNA1. Он получает изменяемое количество Base-аргументов и переправляет эти аргументы в виде последовательности методу fromSeq.

А вот как работают эти две схемы создания:

```
scala> val xs = List(A, G, T, A)
xs: List[Product with Base] = List(A, G, T, A)

scala> RNA1.fromSeq(xs)
res1: RNA1 = RNA1(A, G, T, A)

scala> val rna1 = RNA1(A, U, G, G, T)
rna1: RNA1 = RNA1(A, U, G, G, T)
```

Подгонка типа результата методов RNA

А вот еще диалоги с интерпретатором, касающиеся абстракции RNA1:

```
scala> rna1.length
res2: Int = 5
```

```
scala> rna1.last
res3: Base = T

scala> rna1.take(3)
res4: IndexedSeq[Base] = Vector(A, U, G)
```

Первые два результата имеют вполне ожидаемый характер, но вот последний результат получения первых трех элементов `rna1` отнести к таковым нельзя. Фактически в качестве статического типа результата показан `IndexedSeq[Base]`, а в качестве динамического типа полученного значения показан `Vector`. Можно было ожидать увидеть вместо этого значение типа `RNA1`. Но это невозможно, поскольку код листинга 25.6 лишь делает `RNA1` расширением `IndexedSeq`. В то же время класс `IndexedSeq` располагает методом `take`, который возвращает `IndexedSeq`, а он реализован относительно исходной конструкции `IndexedSeq`, то есть относительно типа `Vector`.

Разобравшись в ситуации, можно задать следующий вопрос: а как ее изменить? Один из способов предусматривает переопределение метода `take` в классе `RNA1`, возможно, таким вот образом:

```
def take(count: Int): RNA1 = RNA1.fromSeq(super.take(count))
```

Задача для `take` будет решена. А как же насчет `drop`, или `filter`, или `init`? Ведь существует более 50 методов работы с последовательностями, возвращающими опять же последовательности. Если быть последовательным, то нужно переопределить все эти методы. Такой вариант все больше теряет свою привлекательность.

К счастью, есть гораздо более простой способ достижения того же эффекта. Нужно, чтобы класс `RNA` стал наследником не только

класса IndexedSeq, но и его трейта с реализациями IndexedSeqLike. Этот вариант показан в листинге 25.7. Новая реализация отличается от предыдущей только двумя аспектами. Во-первых, класс RNA2 теперь также примыкает к IndexedSeqLike[Base, RNA2]. В трейте IndexedSeqLike расширяемым образом реализуются все конкретные методы для IndexedSeq.

Листинг 25.7. Класс РНК-нитей, вторая версия

```
final class RNA2 private (
  val groups: Array[Int],
  val length: Int
)       extends       IndexedSeq[Base]       with
IndexedSeqLike[Base, RNA2] {
  import RNA2._
  override def newBuilder: Builder[Base, RNA2] =
    new ArrayBuffer[Base] mapResult fromSeq
  def apply(idx: Int): Base = // as before
}
```

Например, тип возвращаемого значения таких методов, как take, drop, filter или init (то есть RNA2 в листинге 25.7), передается в качестве второго параметра классу IndexedSeqLike. Чтобы добиться этого, сам трейт IndexedSeqLike основывается на абстракции newBuilder, которая создает построитель нужного вида. Подклассы трейта IndexedSeqLike должны переопределить newBuilder для возвращения коллекций своих собственных видов. В классе RNA2 метод newBuilder возвращает построитель типа Builder[Base, RNA2]. Для создания этого построителя в нем сначала создается ArrayBuffer, который сам относится к типу Builder[Base, ArrayBuffer]. Затем ArrayBuffer преобразуется путем вызова своего метода mapResult в RNA2-

построитель. Метод mapResult ожидает в качестве своего параметра функцию преобразования из ArrayBuffer в RNA2. Данной функцией является просто RNA2.fromSeq, которая преобразует последовательность по произвольному основанию в RNA2-значение (следует напомнить, что буфер массива является разновидностью последовательности, поэтому к нему также можно применить RNA2.fromSeq).

Если определение newBuilder не будет указано, это приведет к выдаче сообщения об ошибке:

```
RNA2.scala:5: error: overriding method newBuilder
in trait
TraversableLike of type =>
scala.collection.mutable.Builder[Base,RNA2];
          method newBuilder in trait
GenericTraversableTemplate of type
                                                =>
scala.collection.mutable.Builder[Base,IndexedSeq[Ba
has
  incompatible type
class RNA2 private (val groups: Array[Int], val
length: Int)
      ^
one error found
```

Это довольно длинное и сложное сообщение, отображающее запутанность способа объединения библиотек коллекций. Информацию о том, откуда берутся методы, лучше проигнорировать, поскольку в данном случае она вряд ли принесет какую-то пользу. Остается принять во внимание, что нужно определить метод newBuilder с типом результата Builder[Base, RNA2], но найден метод newBuilder с типом результата Builder[Base,IndexedSeq[Base]], и он не переопределяет ранее упомянутый метод.

Первый метод с типом результата `Builder[Base, RNA2]` является абстрактным методом, который в листинге 25.7 получает экземпляр этого типа путем передачи параметра типа `RNA2` в адрес `IndexedSeqLike`. Второй метод с типом результата `Builder[Base,IndexedSeq[Base]]` предоставляется унаследованным классом `IndexedSeq`. Иными словами, класс `RNA2` недействителен без определения `newBuilder` с первым типом результата.

Теперь с улучшенной реализацией класса `RNA` в листинге 25.7 такие методы, как `take`, `drop` или `filter`, работают подобающим образом:

```
scala> val rna2 = RNA2(A, U, G, G, T)
rna2: RNA2 = RNA2(A, U, G, G, T)

scala> rna2 take 3
res5: RNA2 = RNA2(A, U, G)

scala> rna2 filter (U !=)
res6: RNA2 = RNA2(A, G, G, T)
```

Работа с методом map и родственными ему по свойствам методами

В коллекциях есть еще один класс методов, который нам пока не встречался. Эти методы не всегда возвращают именно тот же тип коллекции. Они могут возвращать ту же разновидность коллекции, но с другим типом элементов. Классическим примером может послужить метод `map`. Если `s` относится к типу `Seq[Int]`, а `f` является функцией преобразования `Int` в `String`, то выражение `s.map(f)` будет возвращать значение типа `Seq[String]`. Таким образом, тип элемента изменился от полученного типа до типа результата, но разновидность коллекции осталась той же самой.

Есть и другие методы, которые ведут себя так же, как `map`. От

некоторых из них этого вполне можно было ожидать (например, от `flatMap`, `collect`), а вот в отношении других такое поведение может стать неожиданностью. Например, метод добавления `++` также может выдать результат, чей тип отличается от типа аргументов метода, — добавление списка `String`-элементов к списку `Int`-элементов приведет к выдаче списка `Any`-элементов. Но как эти методы можно приспособить к работе с РНК-нитями? В идеале ожидается, что отображение оснований на основания РНК-нити выдаст снова РНК-нить:

```
scala> val rna = RNA(A, U, G, G, T)
rna: RNA = RNA(A, U, G, G, T)

scala> rna map { case A => T case b => b }
res7: RNA = RNA(T, U, G, G, T)
```

Аналогично этому сложение двух РНК-нитей с применением оператора `++` должно возвращать снова РНК-нить:

```
scala> rna ++ rna
res8: RNA = RNA(A, U, G, G, T, A, U, G, G, T)
```

В то же время отображение оснований на какой-либо другой тип РНК-нити не может выдать другую РНК-нить, поскольку у нового элемента будет неподходящий тип. Вместо этого отображение должно выдать последовательность. По аналогии с этим добавление к РНК-нити элементов, не относящихся к типу оснований, то есть к `Base`, может выдать обобщенную последовательность, но не может выдать другую РНК-нить:

```
scala> rna map Base.toInt
res2: IndexedSeq[Int] = Vector(0, 3, 2, 2, 1)

scala> rna ++ List("missing", "data")
res3: IndexedSeq[java.lang.Object] =
```

```
  Vector(A, U, G, G, T, missing, data)
```

Это то, что ожидается в идеале. Но это совсем не то, что предоставляется классом RNA2 из листинга 25.7. Фактически, если запустить показанные ранее первые два примера с экземплярами этого класса, будут получены следующие результаты:

```
scala> val rna2 = RNA2(A, U, G, G, T)
rna2: RNA2 = RNA2(A, U, G, G, T)

scala> rna2 map { case A => T case b => b }
res0: IndexedSeq[Base] = Vector(T, U, G, G, T)

scala> rna2 ++ rna2
res1: IndexedSeq[Base] = Vector(A, U, G, G, T, A,
U, G, G, T)
```

Получается, что результатом map и ++ никогда не будет РНК-нить, даже если типом элемента создаваемой коллекции является Base. Чтобы понять, как можно исправить ситуацию, есть смысл присмотреться к сигнатуре метода map (или метода ++, имеющего сходную сигнатуру). Изначально метод map был определен в классе scala.collection.TraversableLike со следующей сигнатурой:

```
def map[B, That](f: Elem => B)
    (implicit cbf: CanBuildFrom[Repr, B, That]):
That
```

Здесь Elem является типом элементов коллекции, а Repr — типом самой коллекции, то есть вторым параметром типа, который передается таким классам реализации, как TraversableLike и IndexedSeqLike. Метод map получает два дополнительных параметра, B и That. Параметр B означает тип результата функции отображения, а также является типом элементов новой коллекции. Параметр That появляется в качестве

типа результата выполнения метода map, таким образом, он представляет тип новой создаваемой коллекции.

Как определяется тип That? Он связан с другими типами подразумеваемым параметром cbf, имеющим тип CanBuildFrom[Repr, B, That]. Такие подразумеваемые CanBuildFrom определяются отдельными классами коллекций. По сути, подразумеваемое значение типа CanBuildFrom[From, Elem, To] говорит: «Это способ, позволяющий при предоставлении коллекции типа From создать из элементов типа Elem коллекцию типа To».

Теперь поведение map и ++ в отношении последовательностей RNA2 проясняется. Там нет экземпляра CanBuildFrom, создающего RNA2-последовательности, поэтому в объекте-спутнике унаследованного трейта IndexedSeq был найден следующий наиболее подходящий CanBuildFrom. Благодаря этой подразумеваемости были созданы последовательности типа IndexedSeq, что, собственно, и было показано при применении map к rna2.

Для устранения данного недостатка вам нужно определить в объекте-спутнике класса RNA подразумеваемый экземпляр CanBuildFrom. У этого экземпляра должен быть тип CanBuildFrom[RNA, Base, RNA]. Поскольку в данном экземпляре утверждается, что при получении РНК-нити (RNA) и нового типа элементов, Base, можно создать еще одну коллекцию, которая также будет РНК-нитью (RNA). Подробности показаны в листингах 25.8 и 25.9.

Листинг 25.8. Класс РНК-нитей, окончательная версия

```
final class RNA private (val groups: Array[Int],
val length: Int)
           extends     IndexedSeq[Base]     with
```

```
IndexedSeqLike[Base, RNA] {

  import RNA._

    // Обязательное создание новой реализации
'newBuilder' в 'IndexedSeq'
     override protected[this] def newBuilder:
Builder[Base, RNA] =
    RNA.newBuilder

      // Обязательная реализация 'apply' в
'IndexedSeq'
  def apply(idx: Int): Base = {
    if (idx < 0 || length <= idx)
      throw new IndexOutOfBoundsException
     Base.fromInt(groups(idx / N) >> (idx % N * S)
& M)
  }

    // Необязательное создание новой реализации
foreach
  // для повышения эффективности работы.
  override def foreach[U](f: Base => U): Unit = {
    var i = 0
    var b = 0
    while (i < length) {
      b = if (i % N == 0) groups(i / N) else b >>>
S
      f(Base.fromInt(b & M))
      i += 1
    }
  }
}
```

Листинг 25.9. Объект-спутник RNA, окончательная версия

```
object RNA {
   private val S = 2                // Количество
разрядов в группе
   private val M = (1 << S) - 1 // битовая маска
для изоляции группы
  private val N = 32 / S        // количество групп
в Int-значении

  def fromSeq(buf: Seq[Base]): RNA = {
     val groups = new Array[Int]((buf.length + N -
1) / N)
    for (i <- 0 until buf.length)
       groups(i / N) |= Base.toInt(buf(i)) << (i %
N * S)
    new RNA(groups, buf.length)
  }
  def apply(bases: Base * ) = fromSeq(bases)
  def newBuilder: Builder[Base, RNA] =
    new ArrayBuffer mapResult fromSeq
    implicit  def  canBuildFrom:  CanBuildFrom[RNA,
Base, RNA] =
    new CanBuildFrom[RNA, Base, RNA] {
      def apply(): Builder[Base, RNA] = newBuilder
       def apply(from: RNA): Builder[Base, RNA] =
newBuilder
    }
}
```

Если сравнивать с классом RNA2, здесь имеются два важных отличия. Во-первых, реализация newBuilder перемещена из класса RNA в его объект-спутник. Новый метод newBuilder в

классе RNA просто выполняет перенаправление на это определение. Во-вторых, теперь в объекте RNA имеется подразумеваемое значение CanBuildFrom. Чтобы добавить соответствующий объект, следует в трейте CanBuildFrom определить два метода apply. Оба они создают новый построитель для RNA-коллекции, но имеют разные списки аргументов.

Метод apply() просто создает новый построитель нужного типа. В отличие от него, метод apply(from) получает в качестве аргумента исходную коллекцию. Это может пригодиться при подгонке динамического типа значения, возвращаемого построителем, под динамический тип получателя. В случае с RNA это не будет использоваться, поскольку RNA является терминальным классом, поэтому получатель статического типа RNA также имеет RNA в качестве своего динамического типа. Так что apply(from) также просто вызывает newBuilder, игнорируя свои аргументы.

Вот и все. В классе RNA из листинга 25.8 реализуются все методы работы с коллекциями с использованием их естественных типов. Их реализация требует немного более строгого соблюдения рабочих стандартов. Проще говоря, нужно знать, где помещать фабрики newBuilder и подразумеваемые элементы canBuildFrom. Положительно то, что при относительно небольшом объеме кода будет получено большое количество автоматически определяемых методов. К тому же, если нет намерений выполнять над вашей собственной коллекцией такие массовые операции, как take, drop, map или ++, можно выбрать вариант отказа от дополнительного увеличения объема кода и остановиться на реализации, показанной в листинге 25.6.

До сих пор разговор шел о минимальном количестве определений, необходимых для определения новых последовательностей с методами, подчиняющимися конкретным типам. Но на практике вашим последовательностям может понадобиться также добавление новых функциональных

возможностей или переопределение для повышения эффективности существующих методов. Примером может послужить переопределение метода `foreach` в классе `RNA`. Метод `foreach` сам по себе играет важную роль, поскольку он реализует циклы в отношении коллекций. Более того, на основе `foreach` действуют многие другие методы работы с коллекциями. Поэтому имеет смысл приложить усилия к оптимизации реализации этого метода.

Стандартная реализация `foreach` в `IndexedSeq` станет просто выбирать каждый `i`-й элемент коллекции с помощью `apply`, где `i` будет относиться к диапазону от нуля до длины коллекции минус единица. То есть эта стандартная реализация для каждого элемента РНК-нити выбирает элемент массива и распаковывает из него основание.

Переопределение `foreach` в классе `RNA` ведет себя рациональнее. Заданная функция тут же применяется ко всем содержащимся в выбранном элементе массива основаниям. Таким образом затраты на выбор из массива и распаковку битов существенно сокращаются.

Внедрение новых наборов и отображений

В качестве второго примера вы освоите способ внедрения в среду коллекций новой разновидности отображения. Замысел заключается в реализации изменяемого отображения с `String` в качестве типа ключей дерева Patricia[129]. Термин *Patricia* является аббревиатурой от Practical Algorithm to Retrieve Information Coded in Alphanumeric (практический алгоритм получения информации, закодированной алфавитно-цифровыми символами). Идея состоит в хранении набора или отображения в качестве дерева, где последующие символы в ключе поиска уникальным образом определяют дерево потомков.

Например, дерево Patricia, в котором хранятся пять строк: «abc»,

«abd», «al», «all», «xy», будет похоже на дерево, показанное на рис. 25.1. Чтобы в этом дереве найти узел, соответствующий строке «abc», нужно просто проследовать к поддереву с меткой «a», перейти оттуда к поддереву с меткой «b», чтобы в конечном счете добраться до поддерева, имеющего метку «c». Если дерево Patricia используется в качестве отображения, значение, связанное с ключом, сохраняется в узлах, добраться до которых можно будет с помощью ключа. Если это набор, просто сохраняется метка, сообщающая, что узел присутствует в наборе.

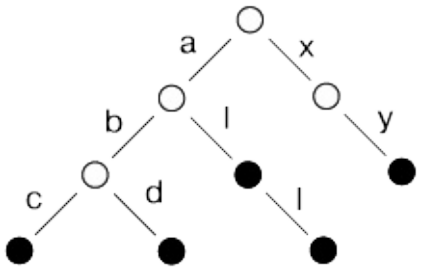


Рис. 25.1. Пример дерева Patricia

В деревьях Patricia поддерживаются весьма эффективные способы поиска и обновления. Еще одной приятной особенностью является то, что они поддерживают выбор подколлекции по заданному префиксу. Например, в дереве, изображенном на рис. 25.1, можно получить подколлекцию всех ключей, начинающихся с «a», просто поставив после корня дерева ссылку «a».

Теперь, основываясь на этих идеях, мы проведем вас через реализацию отображения, выполненного как дерево Patricia. Отображение назовем `PrefixMap`, что будет означать: в нем предоставляется метод `withPrefix`, выбирающий подотображение всех ключей, начинающихся с заданного префикса.

Сначала определим префиксное отображение с ключами, показанными на рис. 25.1:

```
scala> val m = PrefixMap("abc" -> 0, "abd" -> 1,
```

```
"al" -> 2,
  "all" -> 3, "xy" -> 4)
m: PrefixMap[Int] = Map((abc,0), (abd,1), (al,2),
(all,3),
  (xy,4))
```

Затем вызов `withPrefix` в отношении `m` выдаст еще одно префиксное отображение:

```
scala> m withPrefix "a"
res14: PrefixMap[Int] = Map((bc,0), (bd,1), (l,2),
(ll,3))
```

Определение `PrefixMap` показано в листинге 25.10. Этот класс параметризуется типом связанных значений `T` и расширяет `mutable.Map[String, T]` и `mutable.MapLike[String, T, PrefixMap[T]]`. Такая схема уже использовалась для последовательностей в примере с РНК-нитями. А теперь наследование реализации такого класса, как `MapLike`, служит для получения правильного типа результата для таких преобразований, как `filter`.

Листинг 25.10. Реализация префиксных отображений с помощью деревьев Patricia

```
import collection._

class PrefixMap[T]
extends mutable.Map[String, T]
   with mutable.MapLike[String, T, PrefixMap[T]] {
   var suffixes: immutable.Map[Char, PrefixMap[T]]
= Map.empty
  var value: Option[T] = None
```

```
  def get(s: String): Option[T] =
    if (s.isEmpty) value
      else suffixes get (s(0)) flatMap (_.get(s
substring 1))

  def withPrefix(s: String): PrefixMap[T] =
    if (s.isEmpty) this
    else {
      val leading = s(0)
      suffixes get leading match {
        case None =>
          suffixes = suffixes + (leading -> empty)
        case _ =>
      }
      suffixes(leading) withPrefix (s substring 1)
    }
  override def update(s: String, elem: T) =
    withPrefix(s).value = Some(elem)
  override def remove(s: String): Option[T] =
     if (s.isEmpty) { val prev = value; value =
None; prev }
     else suffixes get (s(0)) flatMap (_.remove(s
substring 1))
  def iterator: Iterator[(String, T)] =
    (for (v <- value.iterator) yield ("", v)) ++
    (for ((chr, m) <- suffixes.iterator;
           (s, v) <- m.iterator) yield (chr +: s,
v))
    def += (kv: (String, T)): this.type = {
update(kv._1, kv._2); this }
   def -= (s: String): this.type = { remove(s);
this }
  override def empty = new PrefixMap[T]
```

```
}
```

Узел префиксного отображения имеет два изменяемых поля, `suffixes` и `value`. Поле `value` содержит связанное с этим узлом значение, которое может и отсутствовать. Оно инициализировано в `None`. Поле `suffixes` содержит отображение от символов на `PrefixMap`-значения. Оно инициализируется в пустое отображение. Может возникнуть вопрос: почему в качестве типа реализации `suffixes` было выбрано неизменяемое отображение? Не будет ли более подходящим изменяемое отображение, поскольку `PrefixMap` в целом является изменяемым? Дело в том, что работа с неизменяемыми отображениями, содержащими всего несколько элементов, очень эффективна как по расходу пространства памяти, так и по времени выполнения.

Например, отображения, содержащие менее пяти элементов, представляются в виде отдельного объекта. В отличие от этого, как говорилось в разделе 17.2, стандартным изменяемым отображением является `HashMap`, которое обычно занимает 80 байтов памяти, даже если оно пустое. Следовательно, когда чаще всего используются небольшие коллекции, лучше делать выбор в пользу неизменяемых отображений. В случае с деревом Patricia мы ожидаем, что большинство узлов, за исключением тех, что находятся на самой верхушке дерева, будет иметь совсем немного потомков. Хранение этих потомков в неизменяемом отображении окажется, скорее всего, более эффективным.

Теперь посмотрим на первый метод, который нужно реализовать для работы с отображением, — `get`. Здесь используется следующий алгоритм: для получения значения, связанного в префиксном отображении с пустой строкой, нужно просто выбрать сохраненное в корне дерева значение, которое может и отсутствовать. В противном случае, если строка ключа не пуста, нужно попробовать выбрать подотображение, соответствующее первому символу строки. Если затем будет выдано отображение, нужно продолжить поиск по тому, что

останется в строке ключей этого отображения после ее первого символа. Если выбор окажется неудачным, ключ не хранится в отображении и возвращается значение None. Комбинированный выбор в отношении значения, которое может отсутствовать, можно вполне изящно выразить с помощью flatMap. При применении к значению, которое может отсутствовать, ov и указании замыкания f (оно в свою очередь возвращает значение, которое может отсутствовать) выражение ov flatMap f будет успешно вычислено, если и ov, и f вернут вполне определенное значение. В противном случае выражение ov flatMap f возвратит None.

Следующими двумя методами, реализуемыми для работы с изменяемым отображением, будут += и -=. В реализации, показанной в листинге 25.10, они определены относительно двух других методов, update и remove. Метод remove очень похож на get, за исключением того, что перед возвращением любого связанного с ним значения для поля, содержащего это значение, устанавливается значение None. Метод update сначала вызывает withPrefix для перехода по дереву к тому узлу, который нуждается в обновлении, а затем устанавливает для поля value этого узла заданное значение. Метод withPrefix выполняет переход по дереву, создавая при необходимости поддеревья, если какой-то префикс символов еще не содержится в дереве в качестве пути.

Последним абстрактным методом, реализуемым для изменяемого отображения, является iterator. Он требует создания итератора, выдающего все пары «ключ — значение», хранящиеся в отображении. Для любого заданного префиксного отображения этот итератор состоит из следующих частей: если отображение содержит в поле value своего корня определенное значение Some(x), то первым элементом, возвращенным из итератора, будет ("", x). Более того, итератор нуждается в проходе итераторов всех подотображений, сохраненных в поле suffixes, при этом нужно добавить символ перед каждой строкой

ключа, возвращенной этими итераторами. Точнее говоря, если `m` является подотображением, полученным из корня посредством символа `chr`, а `(s, v)` — элементом, возвращенным итератором `m.iterator`, то итератор корня возвратит `(chr +: s, v)`.

Эта логика в реализации метода `iterator`, показанной в листинге 25.10, проявляется в виде объединения двух выражений `for`. Первое выражение `for` выполняет обход `value.iterator`. Здесь используется то обстоятельство, что в `Option`-значениях определяется метод итерации, который либо вообще не возвращает элемент, если для значения, которое может отсутствовать, используется `None`, либо возвращает ровно один элемент `x`, если значение, которое может отсутствовать, представлено как `Some(x)`.

Обратите внимание на то, что никакого нового метода `newBuilder` в `PrefixMap` не определено. В этом нет необходимости, поскольку отображения и наборы располагают исходными построителями, являющимися экземплярами класса `MapBuilder`. Для изменяемого отображения исходный построитель начинает с пустого отображения, а затем добавляет один за другим элементы с использованием имеющегося в отображении метода `+=`. Изменяемые наборы ведут себя точно так же. Исходные построители для неизменяемых отображений и наборов используют вместо метода `+=` неразрушающий метод добавления элементов `+`. Но во всех этих случаях, чтобы создать правильную разновидность набора или отображения, нужно начинать с пустого набора или отображения этого вида. Это обеспечивается методом `empty`, который является последним методом, определяемым в `PrefixMap`. В листинге 25.10 этот метод просто возвращает свежий `PrefixMap`-объект.

Теперь обратимся к объекту-спутнику `PrefixMap`, показанному в листинге 25.11. На самом деле абсолютной необходимости в определении этого объекта-спутника не было, поскольку класс `PrefixMap` сам в состоянии справиться со всеми задачами.

Главным предназначением объекта PrefixMap является определение некоторых удобных фабричных методов. В нем также определяется подразумеваемый элемент CanBuildFrom, предназначенный для улучшения работы с типами.

Двумя удобными методами являются empty и apply. В среде коллекций Scala для всех других коллекций имеются точно такие же методы, поэтому был смысл также определить их и в этом объекте.

Листинг 25.11. Объект-спутник для префиксных отображений

```
import scala.collection.mutable.{Builder,
MapBuilder}
import scala.collection.generic.CanBuildFrom

object PrefixMap {
  def empty[T] = new PrefixMap[T]
  def apply[T](kvs: (String, T) * ): PrefixMap[T]
= {
    val m: PrefixMap[T] = empty
    for (kv <- kvs) m += kv
    m   }

    def newBuilder[T]: Builder[(String, T),
PrefixMap[T]] =
    new MapBuilder[String, T, PrefixMap[T]](empty)
  implicit def canBuildFrom[T]
      : CanBuildFrom[PrefixMap[_], (String, T),
PrefixMap[T]] =
      new CanBuildFrom[PrefixMap[_], (String, T),
PrefixMap[T]] {
                def apply(from: PrefixMap[_]) =
newBuilder[T]
```

```
        def apply() = newBuilder[T]
      }
}
```

Располагая этими двумя методами, литералы `PrefixMap` можно записывать точно так же, как это делалось для любых других коллекций:

```
scala> PrefixMap("hello" -> 5, "hi" -> 2)
res0: PrefixMap[Int] = Map((hello,5), (hi,2))

scala> PrefixMap.empty[String]
res2: PrefixMap[String] = Map()
```

Еще одним элементом объекта `PrefixMap` является подразумеваемый экземпляр `CanBuildFrom`. Его предназначение точно такое же, как и у определения `CanBuildFrom` в последнем разделе: заставить методы, подобные `map`, возвращать наиболее подходящий тип. Рассмотрим, к примеру, отображение функции на пары «ключ — значение» `PrefixMap`. Поскольку эта функция производит пары из строковых значений и какого-то иного типа, получаемой коллекцией снова будет `PrefixMap`. Рассмотрим пример:

```
scala> res0 map { case (k, v) => (k + "!", "x" *
v) }
res8: PrefixMap[String] = Map((hello!,xxxxx),
(hi!,xx))
```

Аргумент данной функции получает связку «ключ — значение» префиксного отображения `res0` и производит пару строковых значений. В результате получается отображение типа `PrefixMap`, на этот раз с типом значений `String` вместо `Int`. Без подразумеваемого элемента `canBuildFrom` в `PrefixMap` результатом стало бы не префиксное отображение, а обобщенное

изменяемое отображение.

Краткие выводы

Если требуется полное внедрение в среду нового класса коллекции, следует обратить внимание на следующие моменты.

Нужно решить, какой должна быть коллекция, изменяемой или неизменяемой.

18. Требуется подобрать для коллекции подходящие базовые трейты.

19. Чтобы реализовать большинство операций для работы с коллекцией, нужно указать наследование из подходящих трейтов реализаций.

20. Если требуется, чтобы `map` и подобные ей операции возвращали экземпляры вашего типа коллекции, следует в объекте-спутнике вашего класса предоставить подразумеваемый элемент `CanBuildFrom`.

Резюме

Теперь вы увидели, как создаются коллекции Scala и как можно добавлять новые разновидности коллекций. Поскольку в Scala имеется сильная поддержка абстракций, каждый новый тип коллекции может располагать большим количеством методов, не требуя их повторной реализации.

128 В идеале все должно определяться в одном месте, но есть ряд исключений, где требуется переопределение.

129 Morrison D. R. PATRICIA — Practical Algorithm to Retrieve Information Coded in Alphanumeric. J. ACM, 15(4):514–534, 1968. ISSN 0004-5411. doi:http://doi.acm.org/10.1145/321479.3 21481.

26. Экстракторы

Вы, вероятно, уже привыкли к лаконичному способу разложения и анализа данных с помощью поиска по шаблону. В этой главе будет показано, как придать этой концепции более обобщенный характер. До сих пор шаблоны конструктора были связаны с case-классами. Например, `Some(x)` является вполне допустимым шаблоном, поскольку `Some` относится к case-классу. Иногда может возникать потребность в создании подобных шаблонов без создания связанного с ними case-класса. Вам может потребоваться получить возможность создания собственных разновидностей шаблонов. Именно такая возможность и предоставляется экстракторами. В этой главе рассмотрим, что представляют собой экстракторы и как ими можно воспользоваться для определения шаблонов, отвязанных от представления объекта.

26.1. Пример извлечения адресов электронной почты

Чтобы показать задачу, решаемую с помощью экстракторов, предположим, что нужно проанализировать строки, представляющие адреса электронной почты. Требуется решить, есть в заданной строке адрес электронной почты или нет, и, если такой адрес в ней есть, получить доступ к частям пользователя и домена этого адреса. Традиционный способ решения данной задачи предусматривает использование трех вспомогательных функций:

```
def isEMail(s: String): Boolean
def domain(s: String): String
def user(s: String): String
```

Располагая ими, можно проанализировать заданную строку `s`:

```
if (isEMail(s)) println(user(s) + " AT " +
```

```
domain(s))
else println("not an email address")
```

Этот код работает, но выглядит как-то неуклюже. Более того, если понадобится объединить несколько таких проверок, все станет намного сложнее. Например, может понадобиться найти две смежные строки в списке при условии, что обе они являются адресами электронной почты с одинаковыми пользовательскими частями. Можете сами попробовать воспользоваться ранее определенными функциями доступа, чтобы посмотреть, во что все это выльется.

В главе 15 уже было показано, что для решения таких задач идеально подходит поиск по шаблону. Представим на минуту, что можно сопоставить строку следующему шаблону:

```
EMail(user, domain)
```

Шаблон будет соответствовать, если строка содержит встроенный знак (@). В таком случае она привяжет переменную `user` к той части строки, которая находится перед @, а переменную `domain` — к той ее части, которая находится после этого знака. Если сделать такой шаблон условием, предыдущее выражение может быть прописано более четко:

```
s match {
  case EMail(user, domain) => println(user + " AT
" + domain)
  case _ => println("not an email address")
}
```

Более сложная задача нахождения двух смежных адресов электронной почты с одинаковой пользовательской частью будет сведена к следующему шаблону:

```
ss match {
```

```
  case EMail(u1, d1) :: EMail(u2, d2) :: _ if (u1
== u2) => ...
  ...
}
```

Это читать гораздо удобнее, чем все то, что можно было бы написать с помощью функций доступа. Но проблема в том, что строки не являются case-классами — у них нет представления, соответствующего `EMail(user, domain)`. И тут на помощь приходят экстракторы Scala: они позволяют определять новые шаблоны для существующих типов, где шаблон не должен следовать внутреннему представлению типа.

26.2. Экстракторы

Экстрактор в Scala представляет собой объект, у которого в качестве одного из элементов имеется метод `unapply`. Этот метод предназначен для проверки соответствия строкового значения определенному условию и для разделения этого значения на части. Зачастую объект-экстрактор также определяет метод-двойник `apply` для создания значений, но это необязательное условие. В качестве примера в листинге 26.1 показан объект-экстрактор для адресов электронной почты.

Листинг 26.1. Объект извлечения Email-строк

```
object EMail {
  // Метод вставки (необязательный)
  def apply(user: String, domain: String) = user +
"@" + domain
  // Метод извлечения (обязательный)
     def  unapply(str:  String):  Option[(String,
String)] = {
```

```
    val parts = str split "@"
        if (parts.length == 2) Some(parts(0),
parts(1)) else None
  }
}
```

В этом объекте определены оба метода, и `apply`, и `unapply`. Метод `apply` имеет то же самое предназначение, что и всегда: он превращает `EMail` в объект, который может быть применен к аргументам в круглых скобках точно так же, как применяется метод. Следовательно, чтобы создать строку `John@epfl.ch`, можно воспользоваться записью `EMail("John", "epfl.ch")`. Чтобы прояснить ситуацию, можно также позволить `Email` воспользоваться наследованием из функционального типа Scala:

```
object EMail extends ((String, String) => String)
{ ... }
```

ПРИМЕЧАНИЕ

Та часть предыдущего объявления объекта, которая имеет вид (String, String) => String, означает то же самое, что и выражение Function2[String, String, String], которое объявляет абстрактный метод apply, реализуемый EMail. В результате этого объявления можно будет, к примеру, передать EMail тому методу, который ожидает тип Function2[String, String, String].

Метод `unapply` является именно тем элементом, который превращает `EMail` в экстрактор. В определенном смысле он реверсирует процесс создания, выполняемый `apply`. Если `apply` получает две строки и формирует из них строку адреса электронной почты, то `unapply` получает адрес электронной почты и потенциально возвращает две строки: `user` и `domain`, извлеченные из этого адреса. Но `unapply` должен также

справляться со случаями, когда заданная строка не является адресом электронной почты. Именно поэтому `unapply` в отношении пары строк возвращает `Option`-тип. Результатом реализации этого метода является либо `Some(user, domain)`, если строка `str` представляет собой адрес электронной почты, с заданными частями, обозначающими пользователя и домен[130], либо `None`, если `str` не является адресом электронной почты. Вот несколько примеров:

```
unapply("John@epfl.ch") равняется Some("John",
"epfl.ch")
unapply("John Doe") равняется None
```

Теперь каждый раз, когда поиск по шаблону сталкивается с шаблоном, ссылающимся на объект-экстрактор, он вызывает в отношении выражения выбора принадлежащий экстрактору метод `unapply`. Например, выполнение кода

```
selectorString match { case EMail(user, domain) =>
... }
```

приведет к вызову:

```
EMail.unapply(selectorString)
```

Как было показано ранее, этот вызов `EMail.unapply` приведет к возвращению либо `None`, либо `Some(u, d)`, и для последних значений `u` будет пользовательской частью адреса, а `d` — его доменной частью. В случае возвращения `None` соответствие шаблону определено не будет и система попробует другой шаблон или даст сбой с выдачей исключения `MatchError`. В случае возвращения `Some(u, d)` поиск по шаблону пройдет удачно и его переменные будут привязаны к элементам возвращаемого значения. В предыдущем поиске переменная `user` будет привязана к `u`, а переменная `domain` — к `d`.

В примере поиска по шаблону EMail тип String выражения селектора, selectorString, соответствует типу аргумента unapply (который в примере также имел тип String). Такое случается довольно часто, но не является обязательным. Также было бы возможно применить к селектору соответствие EMail-экстрактора для более общих типов. Например, чтобы определить, является ли произвольное значение x строкой адреса электронной почты, можно воспользоваться следующим кодом:

```
val x: Any = ...
x match { case EMail(user, domain) => ... }
```

С этим кодом система поиска по шаблону сначала проверяет, соответствует ли заданное значение x типу String, то есть типу параметра, принадлежащего EMail метода unapply. Если соответствует, происходит приведение значения к типу String и процесс поиска по шаблону продолжается в обычном режиме. Если не соответствует, поиск по шаблону тут же дает сбой.

Работа имеющегося в объекте Email метода apply называется *инъекцией*, поскольку этот метод получает некие аргументы и выдает элемент заданного набора (в нашем случае набора строк, являющихся адресами электронной почты). А работа метода unapply называется *экстракцией*, так как этот метод получает элемент того же самого набора и извлекает некоторые его части (в нашем случае это подстроки user и domain). Инъекции и экстракции часто группируют вместе в одном объекте, поскольку тогда можно будет воспользоваться именем объекта как для конструктора, так и для работы с шаблоном, чем имитируется соглашение, действующее в отношении поиска по шаблону с применением case-классов. Но в объекте можно также определить экстракцию без соответствующей инъекции. Сам объект называется экстрактором независимо от того, содержит он метод apply или нет.

Если включен метод инъекции, он должен быть антиподом

метода экстракции. Например, следующий вызов:

```
EMail.unapply(EMail.apply(user, domain))
```

должен возвращать:

```
Some(user, domain)
```

то есть точно такую же последовательность аргументов, заключенных в Some. Теперь пойдем в обратном направлении, а именно, как показано в следующем коде, сначала запустим unapply, а затем apply:

```
EMail.unapply(obj) match {
  case Some(u, d) => EMail.apply(u, d)
}
```

В этом коде применение match в отношении obj пройдет успешно, и мы вправе ожидать получения обратно от apply того же самого объекта. Эти два условия антиподов для apply и unapply являются принципами конструирования, достойными подражания. Они не навязываются языком Scala, но рекомендуется придерживаться их при создании собственных экстракторов.

26.3. Шаблоны без переменных или с одной переменной

Метод unapply из предыдущего примера в случае удачного завершения возвращает пару элементов-значений. Его легко сделать более универсальным, подходящим под шаблоны из более чем двух переменных. Чтобы привязать *N* переменных, метод unapply будет возвращать *N*-элементный кортеж, заключенный в оболочку Some.

Но ситуация, при которой шаблон привязывает только одну переменную, рассматривается по-иному. В Scala нет кортежей с одним элементом. Чтобы возвращался только один элемент

шаблона, метод unapply просто заключает в оболочку Some сам элемент. Например, объект-экстрактор, показанный в листинге 26.2, определяет apply и unapply для строк, состоящих из некой подстроки, появляющейся в строковой записи дважды.

Листинг 26.2. Объект извлечения дважды встречающейся строки

```
object Twice {
   def apply(s: String): String = s + s    def
unapply(s: String): Option[String] = {
    val length = s.length / 2
    val half = s.substring(0, length)
      if (half == s.substring(length)) Some(half)
else None
  }
}
```

Существует также вероятность того, что шаблон экстрактора вообще не привязывает никаких переменных. В таком случае соответствующий метод unapply возвращает булево значение true при успешном поиске и false — при сбое. Например, объект-экстрактор, показанный в листинге 26.3, определяет строки, состоящие только из символов в верхнем регистре.

Листинг 26.3. Объект-экстрактор UpperCase

```
object UpperCase {
   def unapply(s: String): Boolean = s.toUpperCase
== s }
```

На этот раз в экстракторе определяется только метод unapply, а метод apply в нем отсутствует. Определять apply нет смысла, поскольку здесь нечего составлять.

Следующая функция userTwiceUpper в своем коде поиска по

шаблону применяет вместе все ранее определенные экстракторы:

```
def userTwiceUpper(s: String) = s match {
  case EMail(Twice(x @ UpperCase()), domain) =>
    "match: " + x + " in domain " + domain
  case _ =>
    "no match"
}
```

Первый шаблон этой функции соответствует строкам, являющимся адресами электронной почты, чья пользовательская часть состоит из двух одинаковых строк с символами в верхнем регистре, например:

```
scala> userTwiceUpper("DIDI@hotmail.com")
res0: String = match: DI in domain hotmail.com

scala> userTwiceUpper("DIDO@hotmail.com")
res1: String = no match

scala> userTwiceUpper("didi@hotmail.com")
res2: String = no match
```

Обратите внимание на то, что `UpperCase` в функции `userTwiceUpper` получает пустой список параметров. Этот список нельзя отбросить, поскольку тогда будет проверяться соответствие объекту `UpperCase`! Обратите также внимание на то, что даже если сам метод `UpperCase()` не привязывает никакую переменную, возможность связать переменную со всем соответствующим ей шаблоном все же имеется. Для этого используется стандартная схема привязки переменных, рассмотренная в разделе 15.2: форма `x @ UpperCase()` связывает переменную `x` с шаблоном, соответствующим `UpperCase()`. Например, в показанном ранее первом вызове `userTwiceUpper` переменная `x` была привязана к

"DI", поскольку это было значение, с которым сравнивался шаблон `UpperCase()`.

26.4. Экстракторы переменного количества аргументов

Все предыдущие методы экстракции для адресов электронной почты возвращали фиксированное количество элементов значений. Иногда такой подход недостаточно гибок. Например, может понадобиться найти соответствие строке, представляющей доменное имя, чтобы каждая часть домена сохранялась в другом подшаблоне. Для этого потребуется выразить шаблоны следующим образом:

```
dom match {
  case Domain("org", "acm") => println("acm.org")
      case Domain("com", "sun", "java") =>
println("java.sun.com")
   case Domain("net", _ * ) => println("a .net
domain")
}
```

В этом примере все устроено так, что домены раскрываются в обратном порядке — от домена верхнего уровня к поддоменам. Сделано это затем, чтобы извлечь больше пользы от последовательно применяемых шаблонов. В разделе 15.2 было показано, что универсальный шаблон сопоставления с последовательностью `_ *` в конце списка аргументов соответствует любым оставшимся в последовательности элементам. От этого свойства больше пользы, если домен верхнего уровня стоит первым, поскольку тогда можно воспользоваться универсальным шаблоном сопоставления с последовательностью для поиска соответствия поддоменам произвольной глубины вложения.

Остается открытым вопрос о том, как экстрактор может

поддерживать *поиск соответствия переменному количеству аргументов*, как показано в предыдущем примере, при том условии, что у шаблонов может быть произвольное количество подшаблонов. Встречавшиеся до сих пор методы `unapply` здесь не подойдут, поскольку каждый из них в случае успешного завершения возвращает фиксированное количество подэлементов. Чтобы справиться с подобными обстоятельствами, Scala позволяет определять другой метод экстракции, специально предназначенный для поиска соответствия переменному количеству аргументов. Этот метод называется `unapplySeq`. Чтобы разобраться в его конструкции, рассмотрим экстрактор `Domain`, показанный в листинге 26.4:

Листинг 26.4. Объект извлечения строк по имени Domain

```
object Domain {
  // Метод вставки (необязательный)
  def apply(parts: String * ): String =
    parts.reverse.mkString(".")
  // Метод извлечения (обязательный)
          def     unapplySeq(whole:     String):
Option[Seq[String]] =
    Some(whole.split("\\.").reverse)
}
```

В объекте `Domain` определяется метод `unapplySeq`, который сначала разбивает строку на части, отделенные друг от друга точками. Это действие выполняется Java-методом `split`, применяемым к строкам, который получает в качестве своего аргумента регулярное выражение. Результатом выполнения метода `split` является массив из подстрок. А результатом выполнения метода `unapplySeq` — массив со всеми элементами, следующими в обратном порядке, заключенный в оболочку `Some`.

Тип результата unapplySeq должен соответствовать Option[Seq[T]], где тип элемента T может быть произвольным. Как было показано в разделе 17.1, Seq является весьма важным классом в иерархии коллекций Scala. Это общий родительский класс для нескольких классов, дающих описания различных видов коллекций: List-коллекций, Array-коллекций, WrappedString-коллекций и некоторых других.

Для симметрии в объекте Domain также имеется метод apply, который выстраивает строку доменного имени из переменного числа параметров частей домена, начиная с имени домена верхнего уровня. Как всегда, наличие метода apply необязательно.

Экстрактор Domain можно использовать для получения более детализированной информации о строках адресов электронной почты. Например, для поиска адресов электронной почты с именем tom в каком-либо домене .com, можно воспользоваться следующей функцией:

```
def isTomInDotCom(s: String): Boolean = s match {
  case EMail("tom", Domain("com", _ * )) => true
  case _ => false
}
```

Этот код даст вполне ожидаемые результаты:

```
scala> isTomInDotCom("tom@sun.com")
res3: Boolean = true

scala> isTomInDotCom("peter@sun.com")
res4: Boolean = false

scala> isTomInDotCom("tom@acm.org")
res5: Boolean = false
```

Можно также наряду с переменной частью получить в качестве

возвращаемого значения unapplySeq ряд фиксированных элементов. Это выражается в возвращении всех элементов в кортеже, где переменная часть, как обычно, будет последней. В примере, показанном в листинге 26.5, демонстрируется новый экстрактор для адресов электронной почты, где доменная часть уже разложена в последовательность.

Листинг 26.5. Объект извлечения ExpandedEMail

```
object ExpandedEMail {
  def unapplySeq(email: String)
      : Option[(String, Seq[String])] = {
    val parts = email split "@"
    if (parts.length == 2)
                                    Some(parts(0),
parts(1).split("\\.").reverse)
    else
      None
  }
}
```

Имеющийся в объекте ExpandedEMail метод unapplySeq возвращает представленное парой (Tuple2) значение, которое может отсутствовать. Первый элемент этой пары является пользовательской частью. А второй элемент — это последовательность имен, представляющих домен. Поиск соответствия этому можно выполнять, как обычно:

```
scala> val s = "tom@support.epfl.ch"
s: String = tom@support.epfl.ch

scala> val ExpandedEMail(name, topdom, subdoms @ _
* ) = s
```

```
name: String = tom
topdom: String = ch
subdoms: Seq[String] = WrappedArray(epfl, support)
```

26.5. Экстракторы и шаблоны последовательностей

В разделе 15.2 было показано, что доступ к элементам списка или массива можно получить с помощью следующих шаблонов последовательностей:

```
List()
List(x, y, _ * )
Array(x, 0, 0, _)
```

Фактически все эти шаблоны последовательностей реализованы с использованием экстракторов из стандартной библиотеки Scala. Например, применение шаблонов вида `List(...)` стало возможным благодаря тому, что объект-спутник `scala.List` является экстрактором, определяющим метод `unapplySeq`. Соответствующие определения показаны в листинге 26.6.

Листинг 26.6. Экстрактор, определяющий метод unapplySeq

```
package scala
object List {
  def apply[T](elems: T * ) = elems.toList
   def unapplySeq[T](x: List[T]): Option[Seq[T]] =
Some(x)
  ...
}
```

В объекте `List` содержится метод `apply`, получающий переменное число аргументов. Это позволяет воспользоваться следующими выражениями:

```
List()
List(1, 2, 3)
```

В нем также содержится метод `unapplySeq`, возвращающий все элементы списка в виде последовательности. Именно он и поддерживает шаблоны вида `List(...)`. Очень похожее определение имеется в объекте `scala.Array`. Им поддерживаются аналогичные инъекции и экстракции в отношении массивов.

26.6. Сравнение экстракторов и case-классов

При всей немалой пользе, получаемой от case-классов, у них имеется один существенный недостаток: с их помощью выявляется конкретное представление данных. Это означает, что имя класса в конструкторском шаблоне соответствует конкретному представлению типа объекта выбора. Если поиск по шаблону вида

```
case C(...)
```

будет выполнен успешно, станет понятно, что выражение селектора является экземпляром класса `C`.

Экстракторы разрывают эту связь между представлением данных и шаблонами. А в примерах этого раздела было показано, что они допускают применение шаблонов, которые не имеют ничего общего с типом данных выбираемого объекта. Это свойство называется *независимостью представлений*. В крупных открытых системах независимость представлений играет весьма важную роль, поскольку позволяет изменять тип реализации, используемый в наборе компонентов, не затрагивая кода тех, кто пользуется этими компонентами.

Если ваш компонент определил и экспортировал набор case-классов, он станет камнем преткновения, поскольку код клиента уже может содержать поиск по шаблону в отношении этих же case-классов. Переименование некоторых case-классов или изменение

иерархии классов повлияет на код клиента. Экстракторы свободны от возникновения подобных проблем, поскольку являются прослойкой, разрывающей прямую связь между представлением данных и их клиентским восприятием.У вас сохраняется возможность изменения конкретного представления типа при условии, что наряду с этим будут обновлены все ваши экстракторы.

Независимость представлений дает экстракторам существенные преимущества над case-классами. Но и у case-классов имеются некоторые преимущества над экстракторами. Во-первых, их намного проще настраивать и определять и для них требуется меньше кода. Во-вторых, они зачастую приводят к более эффективной организации поиска по шаблону, чем экстракторы, поскольку компилятор Scala может оптимизировать шаблоны, применяемые к case-классам, намного лучше шаблонов, применяемых к экстракторам. Дело в том, что механизмы case-классов являются фиксированными, а методы `unapply` или `unapplySeq` в экстракторе способны практически на все. В-третьих, если ваши case-классы являются наследниками запечатанных базовых классов, компилятор Scala проверит ваши поиски по шаблону на исчерпаемость и станет возражать, если какие-либо комбинации возможных значений не были охвачены шаблоном. Для экстракторов любые проверки на исчерпаемость недоступны.

Так какому из двух методов поиска по шаблону следует отдать предпочтение? Все зависит от конкретных обстоятельств. Если создается код для закрытого приложения, то предпочтение обычно отдается case-классам из-за обеспечиваемых ими преимуществ в краткости, скорости и возможности статической проверки кода. Если позже будет принято решение об изменении иерархии классов, приложение нужно будет реструктурировать, но обычно это не вызывает серьезных проблем. В то же время, если требуется распространить тип среди неизвестных вам клиентов, предпочтительным может стать применение экстракторов, поскольку они обеспечивают независимость представлений.

К счастью, немедленно принимать решение необязательно. Всегда можно начать с применения case-классов, а потом при необходимости заменить соответствующий код экстракторами. Поскольку шаблоны для работы с case-классами и шаблоны для работы с экстракторами выглядят в Scala совершенно одинаково, поиск по шаблону в коде ваших клиентов не утратит своей работоспособности.

Разумеется, бывают ситуации, когда с самого начала понятно, что структура шаблонов не соответствует представлению типов ваших данных. Примером могут послужить рассмотренные в данной главе адреса электронной почты. В таком случае единственным доступным вариантом останутся экстракторы.

26.7. Регулярные выражения

Одной из полезных сфер применения экстракторов являются регулярные выражения. Как и в Java, в библиотеке Scala имеются регулярные выражения, но экстракторы делают работу с ними намного привлекательнее.

Составление регулярных выражений

Синтаксис, используемый в Scala для регулярных выражений, унаследован от Java, который в свою очередь унаследовал большинство свойств у Perl. Предполагаем, что этот синтаксис вам уже известен, но если это не так, имеется множество доступных руководств, начиная с документации Javadoc, с описанием класса `java.util.regex.Pattern`. Приведем несколько примеров, которых будет достаточно для освежения памяти.

ab?	Возможно, за a следует b
\d+	Число, состоящее из одной или нескольких цифр, представленных \d
[a-dA-D]\w*	Слово, начинающееся с буквы в диапазоне от a до d в верхнем или нижнем регистре, за которым идет последовательность из нуля и более словообразующих символов, обозначенных \w (к таким символам относятся буква, цифра или знак подчеркивания)

(-)?(\d+)(\.\d *)?	Число, состоящее из необязательного знака «минус», за которым следуют одна или несколько цифр, после которых необязательно стоят точка и от нуля до нескольких цифр. Число состоит из трех групп, то есть из знака «минус», той части, что идет до десятичной точки, и дробной части, включая десятичную точку. Группы заключены в круглые скобки

Класс регулярных выражений находится в пакете `scala.util.matching`:

```
scala> import scala.util.matching.Regex
```

Новое значение регулярного выражения создается передачей строки конструктору `Regex`, например:

```
scala> val Decimal = new Regex("(-)?(\\d+)(\\.\\d
* )?")
Decimal: scala.util.matching.Regex = (-)?(\d+)
(\.\d * )?
```

Обратите внимание на то, что по сравнению с регулярным выражением для десятичных чисел, заданным ранее, в показанной выше строке каждый обратный слеш появляется дважды. Дело в том, что и в Java, и в Scala одиночный обратный слеш является в строковом литерале управляющим символом, а не действующим символом регулярного выражения, показываемым в строке. Поэтому для получения в строке действующего одиночного слеша нужно вместо символа \ использовать пару символов \\.

Если в регулярном выражении содержится множество обратных слешей, его написание и чтение будет затруднено. Альтернативой является имеющаяся в Scala неформатированная строка. Как было показано в разделе 5.2, неформатированной строкой называется последовательность символов, заключенная в тройные кавычки. Разница между неформатированной и простой строкой состоит в том, что неформатированная строка появляется в точности такой, какой она была набрана. Это относится и к обратным слешам, которые не считаются в ней управляющими символами. Таким образом, можно сделать равнозначную и более разборчивую

запись:

```
scala> val Decimal = new Regex("""(-)?(\d+)(\.\d *
)?""")
Decimal:  scala.util.matching.Regex  =  (-)?(\d+)
(\.\d * )?
```

Из выведенной интерпретатором информации видно, что сгенерированное значение результата для `Decimal` точно такое же, что и прежде.

Другой, еще более краткий способ написания регулярного выражения в Scala выглядит следующим образом:

```
scala> val Decimal = """(-)?(\d+)(\.\d * )?""".r
Decimal:  scala.util.matching.Regex  =  (-)?(\d+)
(\.\d * )?
```

Иначе говоря, чтобы получить регулярное выражение, нужно просто добавить к строке пару символов `.r`. Такую возможность обеспечивает наличие в классе `StringOps` метода `r`, который преобразует строку в регулярное выражение. Определение этого метода показано в листинге 26.7.

Листинг 26.7. Определение метода r в классе StringOps

```
package scala.runtime
import scala.util.matching.Regex

class StringOps(self: String) ... {
  ...
  def r = new Regex(self)
}
```

Поиск регулярных выражений

Определить наличие в строке соответствия регулярному выражению можно с помощью нескольких различных операторов:

- `regex findFirstIn str` — поиск первого соответствия регулярному выражению `regex` в строке `str`, результат возвращается в значении `Option`-типа;
- `regex findAllIn str` — поиск всех соответствий регулярному выражению `regex` в строке `str`, результат возвращается в значении типа `Iterator`;
- `regex findPrefixOf str` — поиск соответствия регулярному выражению `regex` в самом начале строки `str`, результат возвращается в значении `Option`-типа.

Например, можно определить входную последовательность, показанную далее, а затем искать в ней десятичные числа:

```
scala> val Decimal = """(-)?(\d+)(\.\d * )?""".r
Decimal: scala.util.matching.Regex = (-)?(\d+)(\.\d * )?

scala> val input = "for -1.0 to 99 by 3"
input: String = for -1.0 to 99 by 3

scala> for (s <- Decimal findAllIn input) println(s)
-1.0
99
3

scala> Decimal findFirstIn input
```

```
res7: Option[String] = Some(-1.0)

scala> Decimal findPrefixOf input
res8: Option[String] = None
```

Извлечение с использованием регулярных выражений

А еще каждое регулярное выражение в Scala определяет экстрактор. Он используется для определения подстрок, соответствующих группам регулярных выражений. Например, строку, обозначающую десятичное число, можно разбить на составляющие следующим образом:

```
scala> val Decimal(sign, integerpart, decimalpart)
= "-1.23"
sign: String = -
integerpart: String = 1
decimalpart: String = .23
```

В этом примере шаблон `Decimal(…)` используется в `val`-определении согласно положениям, рассмотренным в разделе 15.7. Здесь с помощью регулярного выражения определяется метод `unapplySeq`. Он соответствует каждой строке, которая соответствует синтаксису регулярных выражений для десятичных чисел. Если строка соответствует, то в качестве элементов шаблона возвращаются те части, которые отвечают трем группам в регулярном выражении `(-)?(\d+)(\.\d * )?`, которые затем сопоставляются с тремя переменными шаблона: `sign`, `integerpart` и `decimalpart`. Если группа отсутствует, для элемента устанавливается значение `null`, в чем можно убедиться, рассмотрев следующий пример:

```
scala> val Decimal(sign, integerpart, decimalpart)
= "1.0"
sign: String = null
```

```
integerpart: String = 1
decimalpart: String = .0
```

При поиске в for-выражениях экстракторы и регулярные выражения можно смешивать. Например, следующее выражение разбивает на составляющие все десятичные числа, найденные им в строке input:

```
scala> for (Decimal(s, i, d) <- Decimal findAllIn
input)
          println("sign: " + s + ", integer: " +
              i + ", decimal: " + d)
sign: -, integer: 1, decimal: .0
sign: null, integer: 99, decimal: null
sign: null, integer: 3, decimal: null
```

Резюме

В этой главе было показано, как с помощью экстракторов сделать поиск по шаблону более универсальным. Экстракторы позволяют определять собственные разновидности шаблонов, которым не нужно соответствовать типу выбираемых выражений. Это позволяет более гибко подходить к выбору видов шаблонов, допускаемых к использованию в поиске. Фактически это похоже на оперирование разнообразными представлениями одних и тех же данных. Кроме того, вы получаете промежуточный уровень между представлением типов и методом их представления клиентам. Это позволяет выполнять поиск по шаблону, обеспечивая при этом независимость представлений, то есть получить весьма ценное свойство, которое особенно пригодится для больших программных систем.

В вашем арсенале инструментов экстракторы являются еще одним средством, которое позволяет определять весьма гибкие библиотечные абстракции. Они довольно часто используются в

библиотеках Scala, например, чтобы обеспечить возможность удобного поиска с применением регулярных выражений.

130 Как здесь показано, когда Some применяется к кортежу (user, domain), пару круглых скобок при передаче кортежа функции, принимающей один аргумент, можно не ставить. То есть Some(user, domain) означает то же самое, что и Some((user, domain)).

27. Аннотации

Аннотации представляют собой структурированную информацию, добавляемую к исходному коду программы. Как и комментарии, они могут быть разбросаны по всей программе и прикреплены к любым переменной, методу, выражению или другому элементу программы. В отличие от комментариев, у них есть структура, что упрощает процесс их обработки.

В этой главе рассматривается порядок применения аннотаций в Scala: их общий синтаксис и приемы использования ряда стандартных аннотаций.

Создание новых инструментальных средств для обработки комментариев не относится к теме данной книги и здесь не рассматривается. В главе 31 показана одна технология, но она далеко не единственная. Вместо этого основное внимание в данной главе уделяется порядку использования аннотаций, поскольку применять их приходится гораздо чаще, чем определять новые обработчики аннотаций.

27.1. Зачем нужны аннотации

Кроме компиляции и выполнения, с программой можно делать и многое другое. Примерами могут послужить следующие действия.

Автоматическое создание документации, как это делается с применением Scaladoc.

21. Приведение выводимого на стандартное устройство кода в соответствие с предпочитаемым вами стилем.

22. Проверка кода на наличие самых распространенных ошибок, например таких, при которых файл открывается, но при некоторых вариантах хода выполнения программы никогда не

закрывается.

23. Экспериментальная проверка соответствия типов, например для управления побочными эффектами или обеспечения свойств принадлежности.

Такие инструменты называются средствами метапрограммирования, поскольку они программируют то, что другие программы получают в качестве входных данных. Аннотации поддерживают эти инструментальные средства, позволяя программистам расставлять директивы для определенного средства по всему исходному коду. Такие директивы дают возможность средствам работать более эффективно, чем при отсутствии пользовательского ввода. Например, аннотации могут улучшить ранее перечисленный инструментарий следующим образом.

Генератор документации может быть настроен на документирование конкретных методов в качестве устаревших.

24. Средство приведения выводимого на стандартное устройство кода в соответствие предпочтительным стилям может быть настроено на пропуск тех частей программы, которые уже были тщательно отформатированы вручную.

25. Средство проверки незакрытых файлов может быть настроено на игнорирование конкретного файла, закрытие которого уже было проверено программистом.

26. Средство проверки побочных эффектов может быть настроено на проверку отсутствия таковых у указанного метода.

Во всех этих случаях в языке программирования должна иметься теоретическая возможность обеспечения способов вставки

дополнительной информации. Фактически большинство из этого непосредственно поддерживается в тех или иных языках. Но для непосредственной поддержки одним языком такого арсенала инструментальных средств будет многовато. Более того, вся эта информация игнорируется компилятором, от которого требуется лишь сделать код работоспособным.

В подобных случаях философия Scala заключается во включении разумного минимума средств, поддерживаемых независимо друг от друга в основном языке, чтобы можно было воспользоваться широким разнообразием средств метапрограммирования. В данном случае в качестве минимальной поддержки выступает система аннотаций. Компилятору важно только наличие аннотации, но значение каждой конкретной аннотации ему безразлично. Поэтому любое средство метапрограммирования может определять и использовать собственные аннотации.

27.2. Синтаксис аннотаций

Типичное использование аннотации выглядит так:

```
@deprecated def bigMistake() = //...
```

Аннотацией здесь является `@deprecated`, и она применяется ко всему методу `bigMistake`, который для экономии места не показан. В данном случае метод помечен автором `bigMistake` как нечто нежелательное для использования. Возможно, из будущей версии кода `bigMistake` будет полностью изъят.

В предыдущем примере метод аннотирован как `@deprecated`. Аннотации можно применять и в других местах. Допускается использовать их в любых видах объявлений или определений, включая объявления `val`- и `var`-переменных, `def`-определения классов, объектов, трейтов и типов. Аннотация применяется ко всему объявлению или определению, которое за ней следует:

```
@deprecated class QuickAndDirty {
  //...
}
```

Аннотации могут применяться также к выражениям, как в случае с аннотацией @unchecked для поиска по шаблону (см. главу 15). Для этого после выражения ставится двоеточие (:), а затем записывается аннотация. Синтаксически это похоже на использование аннотации в качестве типа:

```
(e: @unchecked) match {
  // неисчерпывающие варианты...
}
```

И наконец, аннотации могут помещаться на типах. Аннотированные типы будут рассмотрены в данной главе чуть позже.

Все показанные до сих пор аннотации выглядели просто как знак «эт», после которого указывался класс аннотации. Такие простые аннотации употребляются довольно часто, но у аннотаций есть и более полная общая форма:

```
@annot(exp1, exp2, ...)
```

Компонент annot указывает на класс аннотации. Его должны включать все аннотации. Компоненты exp являются аргументами аннотации. Для аннотаций вроде @deprecated, которым не нужны никакие аргументы, скобки обычно опускаются, но при желании можно пользоваться записью вида @deprecated(). Аргументы для тех аннотаций, в которых они имеются, следует заключать в круглые скобки, например @serial(1234).

Точная форма аргументов, допустимых для передачи аннотации, зависит от конкретного класса аннотации. Большинство обработчиков аннотаций позволяют предоставлять

только непосредственные константы вроде 123 или "Hello". Но сам компилятор поддерживает произвольные выражения при условии, что они пройдут проверку на соответствие типов.

Некоторые классы аннотаций могут этим воспользоваться, например, чтобы позволить вам ссылаться на другие переменные, находящиеся в области видимости:

```
@cool val normal = "Hello"
@coolerThan(normal) val fonzy = "Heeyyy"
```

Внутренне аннотация в Scala представляется просто как вызов конструктора класса аннотации, при этом знак @ заменяется ключевым словом new и появляется допустимый экземпляр выражения создания. Это означает естественную поддержку поименованных и используемых по умолчанию аргументов, поскольку в Scala уже имеются поименованные и используемые по умолчанию аргументы для вызовов методов и конструкторов. Еще одна сложность касается аннотаций, которые концептуально воспринимают в качестве аргументов другие аннотации, в чем есть потребность у некоторых сред. Записывать аннотацию непосредственно в качестве аргумента к аннотации нельзя, поскольку аннотации не являются допустимыми выражениями. Как показано в следующем диалоге с интерпретатором, в таких случаях следует вместо знака @ использовать ключевое слово new:

```
scala> import annotation._
import annotation._

scala> class strategy(arg: Annotation) extends
Annotation
defined class strategy

scala> class delayed extends Annotation
defined class delayed
```

```
scala> @strategy(@delayed) def f() = {}
<console>:1: error: illegal start of simple expression
       @strategy(@delayed) def f() = {}
                           ^
scala> @strategy(new delayed) def f() = {}
f: ()Unit
```

27.3. Стандартные аннотации

В Scala включен ряд стандартных аннотаций. Они используются для средств с довольно широкой востребованностью и поэтому заслуживают включения в спецификацию языка, но они еще недостаточно устоялись, чтобы заслужить собственный синтаксис. Со временем должен появиться новый набор аннотаций, точно так же добавляемых к стандарту.

Нежелательность

Бывает, что со временем возникает желание избавиться от ранее созданного класса или метода. И, несмотря на имеющуюся возможность, метод может быть вызван кодом, созданным другими людьми. Следовательно, просто удалить метод невозможно, поскольку это приведет к прекращению компилирования кода, созданного сторонними разработчиками.

При нежелательности использования ошибочно созданного метода или класса есть способ постепенно избавиться от него. Такой метод или класс помечается как нежелательный к использованию, после чего все вызывающие его разработчики станут получать предупреждение о нежелательности использования. Им лучше прислушаться к этому предупреждению и обновить свой код! Смысл заключается в том, что по истечении довольно продолжительного времени появляется уверенность, что все значимые клиенты перестали обращаться к нежелательному

методу или классу и поэтому его можно совершенно свободно удалить.

Метод помечается как нежелательный путем установки перед ним аннотации `@deprecated`, например:

```
@deprecated def bigMistake() = //...
```

Такая аннотация заставит компилятор Scala в случае обращения кода к методу выдать предупреждение о нежелательности.

Если в качестве аргумента аннотации `@deprecated` предоставить строку, она будет выдана вместе с предупреждением. Заключенное в эту строку сообщение объяснит разработчикам, что именно им следует использовать взамен нежелательного метода:

```
@deprecated("use newShinyMethod() instead")
def bigMistake() = //...
```

Теперь все обращающиеся к данному методу разработчики получат следующее сообщение:

```
$ scalac -deprecation Deprecation2.scala
Deprecation2.scala:33: warning: method bigMistake in object
Deprecation2 is deprecated: use newShinyMethod() instead
    bigMistake()
    ^
one warning found
```

Изменяемые поля

Программирование многопоточных приложений плохо сочетается с совместно используемым изменяемым состоянием. Поэтому основное внимание при поддержке многопоточного программирования на Scala уделяется передаче сообщений и

сведению к минимуму совместно используемого изменяемого состояния. Подробности рассматриваются в главе 32.

Тем не менее иногда программистам в их многопоточных программах нужно воспользоваться изменяемым состоянием. В таких случаях им поможет аннотация `@volatile`. Она проинформирует компилятор, что рассматриваемая переменная будет использоваться сразу несколькими потоками. Подобные переменные реализуются таким образом, чтобы чтение и запись в них выполнялись медленнее, а обращение из нескольких потоков велось более предсказуемо.

Ключевое слово `@volatile` на разных платформах дает разные гарантии. Но на платформе Java получается такое же поведение, как и при написании поля в коде Java и пометке его Java-модификатором `volatile`.

Двоичная сериализация

Среда для двоичной сериализации включена во многие языки. Среда сериализации помогает превращать объекты в потоки байтов и наоборот. Это пригодится для сохранения объектов на диске или отправки их по сети. Добиться тех же целей помогает применение XML (см. главу 28), но при этом проявляются различные недостатки, касающиеся скорости, использования пространства памяти, гибкости и переносимости.

У Scala нет собственной среды сериализации. Поэтому придется пользоваться средой базовой платформы. А язык Scala предоставляет три аннотации, применяемые для различных сред. Кроме того, компилятор Scala для платформы Java истолковывает эти аннотации тем же способом, что и Java (см. главу 31).

Первая аннотация показывает, поддается ли класс сериализации. Большинство классов пригодно к сериализации, но бывают исключения. К примеру, не поддается сериализации обработка сокета или GUI-окна. По умолчанию класс не считается поддающимся сериализации. Если нужно, чтобы он проходил

сериализацию, к нему следует добавить аннотацию `@serializable`.

Вторая аннотация помогает в работе с подвергаемыми сериализации классами, которые со временем претерпевают какие-либо изменения. К текущей версии класса можно прикрепить серийный номер, добавив аннотацию вида `@SerialVersionUID(1234)`, где вместо 1234 следует поставить выбранный вами серийный номер. Среда должна сохранить этот номер в создаваемом потоке байтов. Если впоследствии такой же поток байтов будет загружен заново и предпринята попытка превращения его в объект, среда сможет проверить, что текущая версия класса имеет тот же серийный номер, что и версия, переданная в потоке байтов. Для внесения в класс несовместимых с прежней сериализацией изменений можно указать другой номер версии. После этого среда автоматически откажется загружать старые экземпляры класса.

И наконец, Scala предоставляет аннотацию `@transient` для тех полей, которые вообще не должны подвергаться сериализации. Если пометить поле аннотацией `@transient`, среда не станет его сохранять, даже если окружающий поле объект был подвергнут сериализации. При загрузке объекта для того поля, тип которого проаннотирован как `@transient`, будет восстановлено исходное значение.

Автоматически создаваемые методы get и set

Обычно код на Scala не нуждается в явном определении для полей методов `get` и `set`, поскольку здесь синтаксис для обращения к полям и для вызова методов примешивается. Но некоторые зависящие от платформы среды ожидают наличия методов `get` и `set`. Для того чтобы решить этот вопрос, в Scala предоставляется аннотация `@scala.reflect.BeanProperty`. Если добавить эту аннотацию к полю, компилятор автоматически сгенерирует для него методы `get` и `set`. Если аннотируется поле по имени `crazy`,

то get-метод будет назван getCrazy, а set-метод — setCrazy.

Сгенерированные методы get и set станут доступными только после завершения прохода компиляции. То есть вызывать эти get- и set-методы из кода, скомпилированного одновременно с аннотированными полями, невозможно. На практике это вряд ли станет проблемой, поскольку в коде на Scala к этим полям можно обращаться напрямую. Данное свойство предназначено для поддержки тех сред, в которых ожидается наличие обычных методов get и set, и, как правило, сама среда и код, которые ею пользуется, в одно и то же время не компилируются.

Tailrec

Аннотация @tailrec обычно добавляется к методу, который должен обладать концевой рекурсией, например из-за ожиданий, что иначе он будет рекурсирован слишком глубоко. Чтобы заставить компилятор Scala выполнить в отношении метода оптимизацию с концевой рекурсией, рассмотренную в разделе 8.9, перед определением метода можно добавить аннотацию @tailrec. Если оптимизацию выполнить невозможно, будет получено предупреждение с объяснением причин.

Unchecked

Аннотация @unchecked интерпретируется компилятором в ходе поиска по шаблону. Благодаря ей компилятор перестает заботиться об отсутствии в выражении соответствия ряда вариантов. Подробности рассмотрены в разделе 15.5.

Методы прямого доступа

Аннотация @native информирует компилятор, что реализация метода предоставляется средой выполнения, а не кодом Scala. Компилятор установит на выходе соответствующие флаги, и

разработчику достаточно будет обеспечить реализацию, используя такие механизмы, как интерфейс прямого доступа к виртуальной машине Java (Java Native Interface (JNI)).

При использовании аннотации `@native` должно быть предоставлено тело метода, но в выход оно внедрено не будет. Например, объявить, что метод `beginCountdown` будет предоставлен средой выполнения, можно следующим образом:

```
@native
def beginCountdown() = {}
```

Резюме

В этой главе рассматривались независимые от применяемой платформы аспекты аннотаций, сведения о которых наиболее востребованы. В первую очередь мы разобрали синтаксис аннотаций, поскольку использовать аннотации приходится гораздо чаще, чем создавать их новые экземпляры. Далее рассматривались вопросы применения ряда аннотаций, поддерживаемых стандартным компилятором Scala, включая `@deprecated`, `@volatile`, `@serializable`, `@BeanProperty`, `@tailrec` и `@unchecked`.

Дополнительная информация об аннотациях применительно к языку Java дается в главе 31. В ней речь идет об аннотациях, нацеленных только на применение Java, раскрывается дополнительное значение аннотаций применительно к Java, рассматриваются вопросы взаимодействия с аннотациями, основанными на Java, и приемы использования Java-механизмов для определения и обработки аннотаций в Scala.

28. Работа с XML

В этой главе рассматривается реализованная в Scala поддержка XML. После общего обзора слабоструктурированных данных в ней показаны основные функциональные возможности Scala по работе с XML: как с помощью литералов XML создаются узлы, как XML сохраняется в файлах и загружается из них и как выполняется разбор XML-узлов с использованием методов запросов и поиска по шаблону. Эта глава является всего лишь кратким введением в возможности работы с XML, но изложенного в ней материала вполне достаточно для того, чтобы начать применять эту технологию на практике.

28.1. Слабоструктурированные данные

XML является формой *слабоструктурированных данных*. Структурирование здесь сильнее, чем у обычных строк, так как содержимое данных сведено в дерево. Но обычный XML менее структурирован, чем объекты языков программирования, поскольку между тегами в нем допускается использование текста в свободной форме, а система типов отсутствует[131]. Слабоструктурированные данные очень хорошо подходят для тех случаев, когда нужно сериализовать (то есть превратить в последовательность битов) программные данные для их сохранения в файле или передачи по сети. Вместо преобразования структурированных данных вплоть до байтов их преобразуют в слабоструктурированные данные, а затем выполняется обратный процесс преобразования. Для преобразований между слабоструктурированными и двоичными данными используются уже существующие библиотечные процедуры, что позволяет сэкономить время для решения более важных задач.

Существует множество форм слабоструктурированных данных, но наиболее широкое распространение в Интернете получил язык

XML. Средства XML имеются в большинстве операционных систем, а доступными библиотеками XML располагают многие языки программирования. Удобство пользования этим языком повышает его востребованность. Большинство средств и библиотек были разработаны в ответ на рост популярности XML, и, скорее всего, разработчики программных систем выберут наряду с другими форматами и XML. Если вы пишете программы, общающиеся через Интернет, то рано или поздно столкнетесь с какими-либо службами, общающимися посредством XML.

Принимая во внимание все эти обстоятельства, для работы с XML в Scala включили специальную поддержку. В текущей главе будет рассмотрена реализованная в Scala поддержка создания XML-кода, его обработки с помощью методов экземпляра класса и обработки с помощью поиска по шаблону. Вдобавок к этому рассматривается общая идиома использования XML в Scala.

28.2. Краткий обзор XML

Структура XML выстраивается из двух базовых элементов — текста и тегов[132]. Текст, как всегда, представлен последовательностью символов. Теги, записываемые как <pod>, состоят из символа «меньше чем», буквенно-цифровой метки и символа «больше чем». Теги могут быть открывающими и закрывающими. Закрывающий тег выглядит практически так же, как и открывающий, за исключением использования слеша перед меткой тега, например </pod>.

Открывающий и закрывающий теги должны соответствовать друг другу, что напоминает применение круглых скобок. За любым открывающим тегом должен в конечном итоге следовать закрывающий тег с той же самой меткой. Поэтому использование следующего кода недопустимо:

```
// Недопустимый код XML
One <pod>, two <pod>, three <pod> zoo
```

Более того, содержимое, находящееся внутри двух соответствующих друг другу тегов, должно быть допустимым кодом XML. В коде не может быть двух пар соответствующих друг другу тегов, накладываемых друг на друга:

```
// Это также недопустимый код
<pod>Three <peas> in the </pod></peas>
```

Но вполне допустима следующая запись:

```
<pod>Three <peas></peas> in the </pod>
```

Из-за принятого правила соответствия тегов структурирование XML заключается в соблюдении вложенности элементов. Элемент формируется парой соответствующих друг другу открывающего и закрывающего тегов, и элементы могут быть вложены друг в друга. В показанном ранее примере весь код `<pod>Three <peas>-</peas> in the </pod>` в целом является элементом, а `<peas></peas>` — вложенным в него элементом.

Это основа основ. Следует знать и о двух других особенностях. Во-первых, существует сокращенная система записи для случая, когда сразу же за открывающим тегом стоит соответствующий ему закрывающий. Нужно просто записать один тег со слешем, помещаемым сразу же после метки тега. Такой тег содержит *пустой элемент*. Для использованного в предыдущем примере пустого элемента может быть применена также сокращенная запись:

```
<pod>Three <peas/> in the </pod>
```

Во-вторых, к открывающему тегу могут быть прикреплены *атрибуты*. Атрибутом называется пара «имя — значение», записанная со знаком равенства между элементами. Имя атрибута является простым неструктурированным текстом, а значение заключается в двойные (`""`) или одинарные (`''`) кавычки.

Выглядят атрибуты следующим образом:

```
<pod peas="3" strings="true"/>
```

28.3. Литералы XML

Scala позволяет набирать код XML в виде литерала везде, где допускается использование выражения. Нужно просто набрать открывающий тег, а затем продолжить набор XML-содержимого. Компилятор перейдет в режим XML-ввода и станет считывать содержимое XML до тех пор, пока ему не попадется закрывающий тег, соответствующий открывающему тегу, с которого начинался литерал XML:

```
scala> <a>
         This is some XML.
         Here is a tag: <atag/>
       </a>
res0: scala.xml.Elem =
<a>
  This is some XML.
  Here is a tag: <atag/>
</a>
```

Результат этого выражения будет относиться к типу `Elem`, что означает: это XML-элемент с меткой «a» и дочерним элементом («Это код XML...» и т. д.). К другим важным XML-классам относятся:

- класс `Node`, являющийся абстрактным родительским классом всех классов XML-узлов;
- класс `Text`, представляющий узел, содержащий простой текст. Например, часть `stuff` литерала `<a>stuff</a>` относится к классу `Text`;

- класс NodeSeq, представляющий последовательность узлов. Многие методы в библиотеке XML работают с NodeSeq-объектами в тех местах, в которых от них можно ожидать обработки отдельно взятых Node-объектов. Но такие методы можно использовать также с отдельными узлами, поскольку Node получается при расширении NodeSeq. Это обстоятельство может показаться немного странным, но в отношении XML все это срабатывает. Отдельно взятый Node-объект можно рассматривать как одноэлементный NodeSeq-объект.

Посимвольно можно записывать какой угодно код XML. Используя в качестве управляющих символов фигурные скобки, в середине XML-литерала можно выполнять вычисления кода Scala. Простой пример вычисления в фигурных скобках выглядит следующим образом:

```
scala> <a> {"hello" + ", world"} </a>
res1: scala.xml.Elem = <a> hello, world </a>
```

Внутри управляющих символов в виде фигурных скобок может помещаться произвольное Scala-содержимое, включая дополнительные XML-литералы. Таким образом, по мере увеличения количества вложений в вашем коде может перемежаться XML-код и обычный код Scala, например:

```
scala> val yearMade = 1955
yearMade: Int = 1955

scala> <a> { if (yearMade < 2000) <old>{yearMade}
</old>
               else xml.NodeSeq.Empty }
       </a>
res2: scala.xml.Elem =
<a> <old>1955</old>
```

```
</a>
```

Если код внутри фигурных скобок вычисляется либо в XML-узел, либо в последовательность XML-узлов, такие узлы вставляются непосредственно в получившемся в результате вычисления виде. Если в показанном ранее примере `yearMade` меньше `2000`, значение `yearMade` заключается в теги `<old>` и все это добавляется к элементу `<a>`. В противном случае не добавляется ничего. Относительно этого примера следует заметить, что «ничего» в качестве XML-узла обозначается `xml.NodeSeq.Empty`.

Выражение внутри управляющих символов в виде фигурных скобок не должно обязательно вычисляться в XML-узел. Оно может вычисляться в любое значение Scala. В таком случае результат преобразуется в строку и вставляется в качестве текстового узла:

```
scala> <a> {3 + 4} </a>
res3: scala.xml.Elem = <a> 7 </a>
```

При выводе узла на стандартное устройство любые встречающиеся в тексте знаки `<`, `>` и `&` будут превращены в управляющие последовательности символов:

```
scala> <a> {"</a>potential security hole<a>"} </a>
res4: scala.xml.Elem = <a> &lt;/a&gt;potential security
hole&lt;a&gt; </a>
```

В отличие от этого, если создавать XML с помощью низкоуровневых строковых операций, можно угодить в следующую ловушку:

```
scala> "<a>" + "</a>potential security hole<a>" +
"</a>"
res5: String = <a></a>potential security hole<a>
</a>
```

Здесь заданная пользователем строка была включена в XML-теги в неизменном виде — в данном случае вместе с и <a>. Этот прием может преподнести программисту исходного кода ряд неприятных сюрпризов, поскольку позволяет пользователю влиять на получающееся в итоге дерево XML за пределами пространства, предоставляемого пользователю внутри элемента <a>. Если конструировать XML только с применением XML-литералов, отказавшись от добавления строк, то всему этому классу проблем можно поставить надежный заслон.

28.4. Сериализация

Разобравшись в достаточной степени в поддержке XML в Scala, можно приступать к написанию первой части сериализатора, превращающей внутренние структуры данных в XML. Для этого понадобятся лишь XML-литералы и их управляющие символы, представленные фигурными скобками.

Предположим, к примеру, что вы реализуете базу данных для систематизации своей обширной коллекции старинных термометров с логотипами Coca-Cola. Чтобы хранить записи каталога, можно создать следующий внутренний класс:

```
abstract class CCTherm {
  val description: String
  val yearMade: Int
  val dateObtained: String
  val bookPrice: Int      // в центах США
  val purchasePrice: Int  // в центах США
  val condition: Int      // от 1 до 10

  override def toString = description
}
```

Это обычный перегруженный данными класс, содержащий различные сведения о том, к примеру, где термометр был изготовлен, когда он был приобретен и сколько за него было заплачено.

Для превращения экземпляров этого класса в XML следует просто добавить метод `toXML`, использующий XML-литералы и управляющие символы в виде фигурных скобок:

```
abstract class CCTherm {
...
def toXML =
  <cctherm>
    <description>{description}</description>
    <yearMade>{yearMade}</yearMade>
    <dateObtained>{dateObtained}</dateObtained>
    <bookPrice>{bookPrice}</bookPrice>
    <purchasePrice>{purchasePrice}</purchasePrice>
    <condition>{condition}</condition>
  </cctherm>
}
```

А вот как выглядит работа этого метода:

```
scala> val therm = new CCTherm {
         val description = "hot dog #5"
         val yearMade = 1952
         val dateObtained = "March 14, 2006"
         val bookPrice = 2199
         val purchasePrice = 500
         val condition = 9
}
therm: CCTherm = hot dog #5

scala> therm.toXML
```

```
res6: scala.xml.Elem =
<cctherm>
                                <description>hot dog
#5</description>
              <yearMade>1952</yearMade>
                                <dateObtained>March 14,
2006</dateObtained>
              <bookPrice>2199</bookPrice>
              <purchasePrice>500</purchasePrice>
              <condition>9</condition>
            </cctherm>
```

ПРИМЕЧАНИЕ

Выражение new CCTherm в примере работает несмотря на то, что CCTherm является абстрактным классом, поскольку этот синтаксис в действительности создает экземпляр безымянного подкласса от класса CCTherm. Безымянные классы рассматривались в разделе 20.5.

Кстати, если в качестве текста XML в код на Scala нужно включить фигурную скобку (`'{'` или `'}'`), вместо того чтобы использовать ее в качестве управляющего символа, нужно просто в одной строке записать две фигурные скобки подряд:

```
scala> <a> {{{{brace yourself!}}}} </a>
res7: scala.xml.Elem = <a> {{brace yourself!}}
</a>
```

28.5. Разборка данных XML-формата на части

Среди множества методов, доступных для работы с экземплярами XML-классов, имеются три, о которых непременно следует

упомянуть. Они позволяют разбирать XML-данные на части, не обращая особого внимания на конкретный способ представления XML в Scala. Эти методы основаны на языке XPath, предназначенном для обработки данных в XML-формате. Как часто бывает при работе с языком Scala, их можно вводить в код на Scala напрямую, не привлекая какого-либо внешнего средства.

- **Извлечение текста.** Весь содержащийся в узле текст, за вычетом любых тегов элемента, можно получить, вызвав в отношении любого XML-узла метод `text`:

```
scala> <a>Sounds <tag/> good</a>.text
res8: String = Sounds good
```

Все закодированные символы автоматически раскодируются:

```
scala> <a> input ---&gt; output </a>.text
res9: String = " input ---> output "
```

- **Извлечение подчиненных элементов.** Если нужно найти подчиненный элемент по имени его тега, следует просто вызвать в отношении этого имени метод `\`:

```
scala> <a><b><c>hello</c></b></a> \ "b"
res10: scala.xml.NodeSeq = NodeSeq(<b><c>hello</c>
</b>)
```

Можно выполнить углубленный поиск и провести его через несколько подчиненных элементов, для чего вместо метода `\` воспользоваться оператором `\\`:

```
scala> <a><b><c>hello</c></b></a> \ "c"
res11: scala.xml.NodeSeq = NodeSeq()
```

```
scala> <a><b><c>hello</c></b></a> \\ "c"
res12: scala.xml.NodeSeq = NodeSeq(<c>hello</c>)

scala> <a><b><c>hello</c></b></a> \ "a"
res13: scala.xml.NodeSeq = NodeSeq()

scala> <a><b><c>hello</c></b></a> \\ "a"
res14: scala.xml.NodeSeq =
NodeSeq(<a><b><c>hello</c></b></a>)
```

ПРИМЕЧАНИЕ

В Scala вместо имеющихся в XPath методов / и // используются методы \ и \\. Дело в том, что с пары символов // в Scala начинаются комментарии! Поэтому пришлось воспользоваться каким-то другим символом и для успешной работы применить другую разновидность слешей.

- **Извлечение атрибутов.** Атрибуты тега можно извлечь, используя все те же методы \ и \\. Просто перед именем атрибута следует поставить знак @:

```
scala> val joe = <employee
          name="Joe"
          rank="code monkey"
          serial="123"/>
joe: scala.xml.Elem = <employee name="Joe"
rank="code monkey"
serial="123"/>

scala> joe \ "@name"
res15: scala.xml.NodeSeq = Joe
```

```
scala> joe \ "@serial"
res16: scala.xml.NodeSeq = 123
```

28.6. Десериализация

Теперь, используя рассмотренные методы разборки XML-данных на части, можно создать вторую часть сериализации, а именно анализатор, вновь переводящий XML-данные в вашу внутреннюю структуру данных. Например, с помощью следующего кода можно выполнить обратный анализ CCTherm-экземпляра:

```
def fromXML(node: scala.xml.Node): CCTherm =
  new CCTherm {
    val description = (node \ "description").text
    val yearMade = (node \ "yearMade").text.toInt
              val dateObtained = (node \
"dateObtained").text
                val bookPrice = (node \
"bookPrice").text.toInt
              val purchasePrice = (node \
"purchasePrice").text.toInt
                val condition = (node \
"condition").text.toInt
  }
```

Этот код просматривает входящий XML-узел по имени node с целью нахождения каждой из шести частей данных, необходимой для определения данных типа CCTherm. Данные, являющиеся текстом, извлекаются с применением метода .text и остаются неизменными. Работа этого метода показана в следующем примере:

```
scala> val node = therm.toXML
```

```
node: scala.xml.Elem =
<cctherm>
                                <description>hot  dog
#5</description>
              <yearMade>1952</yearMade>
                                <dateObtained>March  14,
2006</dateObtained>
              <bookPrice>2199</bookPrice>
              <purchasePrice>500</purchasePrice>
              <condition>9</condition>
            </cctherm>

scala> fromXML(node)
res17: CCTherm = hot dog #5
```

28.7. Загрузка и сохранение

Осталась еще одна, последняя часть, которая нужна для написания сериализатора данных, — преобразование между XML и потоком данных. Реализовать эту часть легче всего, поскольку существуют библиотечные процедуры, которые все сделают за вас. Вам останется лишь вызвать для данных подходящую процедуру.

Для преобразования XML в строку нужно всего лишь вызвать метод `toString`. Экспериментировать с XML в оболочке Scala позволяет наличие работоспособного метода `toString`.

Но лучше все же воспользоваться библиотечной процедурой и полностью преобразовать все в байты. Здесь в получающийся на выходе XML нужно включить директиву, указывающую на используемую кодировку символов. И если кодировка строки в байты выполняется самостоятельно, то ответственность за отслеживание кодировки символов возлагается на вас самих.

Чтобы преобразовать XML в файл, состоящий из байтов, можно воспользоваться командой `XML.save`. Нужно указать имя файла и

сохраняемый в нем узел:

```
scala.xml.XML.save("therm1.xml", node)
```

После запуска показанной команды получившийся в результате этого файл **therm1.xml** примет следующий вид:

```
<?xml version='1.0' encoding='UTF-8'?>
<cctherm>
            <description>hot dog #5</description>
            <yearMade>1952</yearMade>
                        <dateObtained>March  14,
2006</dateObtained>
            <bookPrice>2199</bookPrice>
            <purchasePrice>500</purchasePrice>
            <condition>9</condition>
          </cctherm>
```

Загрузку выполнить проще, чем сохранение, поскольку файл включает все, что нужно знать загрузчику. Нужно всего лишь вызвать метод `XML.loadFile` в отношении имени файла:

```
scala>          val          loadnode          =
xml.XML.loadFile("therm1.xml")
loadnode: scala.xml.Elem =
<cctherm>
     <description>hot dog #5</description>
     <yearMade>1952</yearMade>
     <dateObtained>March 14, 2006</dateObtained>
     <bookPrice>2199</bookPrice>
     <purchasePrice>500</purchasePrice>
     <condition>9</condition>
   </cctherm>
```

```
scala> fromXML(loadnode)
res14: CCTherm = hot dog #5
```

Таковы основные из необходимых вам методов. У этих методов загрузки и сохранения существует множество вариантов, включая методы для чтения и записи в разнообразные системы чтения и записи, входные и выходные потоки.

28.8. Поиск по шаблону в XML

До сих пор мы рассматривали методы разборки XML с помощью метода `text` и XPath-подобных методов `\` и `\\`. Хорошо, когда есть точные сведения о разновидности разбираемой XML-структуры. Но иногда в работе могут находиться сразу несколько из существующих XML-структур. Возможно, это будут несколько разновидностей записей внутри данных, причиной появления которых стало то, что, кроме термометров, вы начали коллекционировать часы и тарелки. Возможно, вам захочется пропустить все пустые места между тегами. Какой бы ни была причина, отфильтровать все возможности поможет поиск по шаблону.

XML-шаблон выглядит практически так же, как и XML-литерал. Основное отличие состоит в том, что при вставке управляющих символов `{}` код внутри `{}` является не выражением, а шаблоном. В шаблоне, заключенном в символы `{}`, может использоваться полноценный язык шаблонов Scala, включая привязку новых переменных, выполнение проверок соответствия типов и игнорирование содержимого с использованием шаблонов `_` и `_ *`. Рассмотрим простой пример:

```
def proc(node: scala.xml.Node): String =
  node match {
    case <a>{contents}</a> => "Это a: " + contents
    case <b>{contents}</b> => "Это b: " + contents
```

```
    case _ => "Это нечто иное."
}
```

В данной функции реализуется поиск по шаблону в трех вариантах. Первый вариант ведет поиск <a>-элемента, чье содержимое состоит из одиночного подчиненного узла. Он привязывает это содержимое к переменной по имени `contents`, а затем вычисляет код, находящийся справа от соответствующей стрелки вправо (=>). Второй вариант делает то же самое, но ведет поиск не <a>-, а <b>-элемента, а третий вариант соответствует всему, что не соответствует любым другим вариантам. Посмотрим на эту функцию в работе:

```
scala> proc(<a>apple</a>)
res18: String = Это a: apple

scala> proc(<b>banana</b>)
res19: String = Это b: banana

scala> proc(<c>cherry</c>)
res20: String = Это нечто иное.
```

Скорее всего, эта функция не отвечает в полной мере вашим потребностям, поскольку она ищет исключительно содержимое, которое находится внутри одиночного подчиненного узла <a> или <b>. Поэтому с поиском соответствия в следующих вариантах она не справится:

```
scala> proc(<a>a <em>red</em> apple</a>)
res21: String = Это нечто иное.

scala> proc(<a/>)
res22: String = Это нечто иное.
```

Если нужно, чтобы функция находила соответствия в вариантах, подобных этому, можно вести поиск соответствия последовательности узлов, а не одному узлу. Шаблон для любой последовательности XML-узлов записывается в виде _ *. Визуально этот шаблон последовательности похож на универсальный шаблон сопоставления (_), за которым следует звезда Клини в стиле регулярных выражений (*). Обновленная функция поиска соответствия не одиночному подчиненному элементу, а последовательности подчиненных элементов выглядит так:

```
def proc(node: scala.xml.Node): String =
  node match {
     case <a>{contents @ _ * }</a> => "Это a: " +
contents
     case <b>{contents @ _ * }</b> => "Это b: " +
contents
    case _ => "Это нечто иное."
}
```

Заметьте, что в результате применения _ * путем использования @-шаблона, рассмотренного в разделе 15.2, выполняется привязка к переменной contents. А вот как новая версия выглядит в работе:

```
scala> proc(<a>a <em>red</em> apple</a>)
res23: String = It's an a: ArrayBuffer(a ,
<em>red</em>,
apple)

scala> proc(<a/>)
res24: String = It's an a: WrappedArray()
```

И заключительный совет: имейте в виду, что XML-шаблоны отлично подходят для выражений в качестве способа обхода

некоторых частей XML-дерева и игнорирования других его частей. Предположим, к примеру, что вам нужно пропустить пустые места между записями в следующей XML-структуре:

```
val catalog =
  <catalog>
    <cctherm>
      <description>hot dog #5</description>
      <yearMade>1952</yearMade>
      <dateObtained>March 14, 2006</dateObtained>
      <bookPrice>2199</bookPrice>
      <purchasePrice>500</purchasePrice>
      <condition>9</condition>
    </cctherm>
    <cctherm>
      <description>Sprite Boy</description>
      <yearMade>1964</yearMade>
      <dateObtained>April 28, 2003</dateObtained>
      <bookPrice>1695</bookPrice>
      <purchasePrice>595</purchasePrice>
      <condition>5</condition>
    </cctherm>
  </catalog>
```

Визуально похоже на то, что внутри элемента `<catalog>` имеются два узла. Но на самом деле их там пять. До, после и между двумя элементами имеются пустые места! Если не принимать их во внимание, можно некорректно обработать записи о термометрах:

```
catalog match {
  case <catalog>{therms @ _ * }</catalog> =>
    for (therm <- therms)
      println("processing: " +
```

```
                (therm \ "description").text)
}
```

```
processing:
processing: hot dog #5
processing:
processing: Sprite Boy
processing:
```

Обратите внимание на все строки, пытающиеся обработать пустые места, как будто они являются настоящими записями о термометрах. В действительности же вам захочется проигнорировать пустые места и обработать только те подчиненные узлы, которые находятся внутри элемента <cctherm>. Этот поднабор можно описать при помощи шаблона <cctherm>{_ * }</cctherm>, и можно ограничить выражение for обходом только тех записей, которые соответствуют этому шаблону:

```
catalog match {
  case <catalog>{therms @ _ * }</catalog> =>
      for (therm @ <cctherm>{_ * }</cctherm> <-
therms)
      println("processing: " +
              (therm \ "description").text)
}
```

```
processing: hot dog #5
processing: Sprite Boy
```

Резюме

В этой главе был дан лишь краткий обзор тех возможностей,

которые предоставляются вам для работы с XML. Существует множество других расширений, библиотек и инструментальных средств, и все они доступны для изучения. Часть из них настроена на работу со Scala, часть разработана для Java, но может применяться и в Scala, а часть нейтральна по отношению к используемым языкам программирования. Целью этой главы было донесение до вас способов использования слабоструктурированных данных для обмена информацией и способов доступа к слабоструктурированным данным посредством имеющейся в Scala поддержки XML.

131 Для XML существуют системы типов, такие как XML Schemas, но в данной книге они не рассматриваются.

132 В действительности все немного сложнее, но для полноценной работы с XML вполне достаточно и этих сведений.

29. Модульное программирование с использованием объектов

В главе 1 утверждалось, что один из способов придания программам на Scala масштабируемости основывается на возможности использования одних и тех же технологий для создания как малых, так и больших программ. До сих пор в книге основное внимание уделялось программированию с прицелом на малое — разработку и реализацию совсем небольших фрагментов программ, на основе которых можно создавать гораздо более объемные программы133. Обратной стороной медали является программирование с прицелом на большое — организацию и сборку небольших фрагментов в более крупные программы, приложения или системы. Мы касались этой темы, когда в главе 13 говорили о пакетах и модификаторах доступа. Если коротко, пакеты и модификаторы доступа позволяют организовать большую программу, используя пакеты в качестве модулей, где под модулем понимается небольшой фрагмент программы с четко определенным интерфейсом и скрытой реализацией.

Хотя само по себе деление программ на пакеты крайне полезно, оно ограничивает возможности, не допуская абстракций. Невозможно перенастроить пакет для работы двумя разными способами в одной и той же программе, а также невозможно организовать наследование между пакетами. Пакет неизменно содержит один абсолютно четкий перечень компонентов, и этот перечень останется фиксированным до тех пор, пока не будет изменен сам код.

В этой главе мы рассмотрим способы использования имеющихся в Scala объектно-ориентированных свойств для повышения модульности программ. Сначала будет показано, как в качестве модуля может применяться простой синглтон-объект. Затем мы рассмотрим способы использования трейтов и классов в

качестве надмодульных абстракций. Эти абстракции в одной и той же программе могут перенастраиваться на работу со множеством модулей множество раз. И наконец, будет показана применимая на практике технология использования трейтов для деления модуля на несколько файлов.

29.1. Суть проблемы

По мере разрастания программы повышается актуальность задачи ее организации в виде модулей. Во-первых, возможность раздельной компиляции различных модулей, составляющих систему, помогает разным командам работать независимо друг от друга. Кроме того, возможность отключать одну реализацию модуля и подключать другую будет полезной потому, что позволит использовать различные конфигурации системы в разных контекстах, например при блочном тестировании на компьютере разработчика, проведении комплексного тестирования, обкатке и развертывании.

Например, вы можете работать с приложением, использующим базу данных и службу сообщений. По мере написания кода может появиться потребность в запуске блочных тестов на том компьютере, где используются версии-имитаторы как базы данных, так и службы сообщений, которые подходящим для тестирования образом имитируют работу этих служб, не требуя обращений по сети к совместно используемым ресурсам. В ходе комплексного тестирования может потребоваться воспользоваться имитатором службы сообщений и вживую обратиться к разработочной базе данных. В ходе обкатки и, конечно же, развертывания в вашей организации предпочтение будет отдано использованию живых версий как базы данных, так и службы сообщений.

Любая технология, чьим предназначением является создание этой разновидности модульности, должна обладать рядом обязательных свойств. Во-первых, иметь конструктор модулей,

обеспечивающий качественное разделение интерфейса и реализации. Во-вторых, быть способным заменить один модуль другим, имеющим точно такой же интерфейс, без внесения изменений или перекомпиляции тех модулей, работа которых зависит от заменяемого модуля. И наконец, иметь возможность соединять модули в единое целое. Задача соединения должна восприниматься как *конфигурирование* системы.

Одним из подходов к решению этой задачи является *внедрение зависимости* (dependency injection), то есть использование технологии на платформе Java с применением таких сред, как Spring и Guice, которые пользуются популярностью у корпоративного сообщества Java[134]. К примеру, Spring позволяет, по сути, представлять интерфейс модуля в виде Java-интерфейса, а реализацию модуля — в виде Java-классов. Можно указывать зависимости между модулями и связывать приложение в единое целое посредством внешних конфигурационных файлов в формате XML. Spring можно использовать со Scala и при этом применять заложенный в Spring подход к достижению модульности ваших Scala-программ на системном уровне, однако, располагая самим языком Scala, вы получаете ряд предоставляемых им альтернативных вариантов. Далее в главе будут показаны способы использования объектов в качестве модулей, позволяющие достичь модульности по-крупному без использования внешних сред.

29.2. Приложение для работы с рецептами

Предположим, вы создаете корпоративное веб-приложение, позволяющее пользователям управлять рецептами, и хотите разложить программные средства по уровням, включая *уровень предметной области* и *уровень приложения*. На уровне предметной области будут определяться *объекты предметной области*, в которые будут заключаться концепции и правила ведения дел, а также будет инкапсулироваться состояние, сохраняемое во внешней реляционной базе данных. На уровне приложения будет

предоставляться API, сведенный к понятиям тех сервисов, которые приложение предлагает клиентам (включая уровень пользовательского интерфейса). Уровень приложения будет нацелен на реализацию этих сервисов путем координации задач и делегирования работы с объектами уровню предметной области[135]. Хотелось бы на каждом из этих уровней иметь возможность подключения реальных версий или версий-имитаторов того или иного объекта, чтобы проще было создавать блочные тесты для вашего приложения. Для воплощения этих замыслов в жизнь объекты, поведение которых нужно имитировать, следует рассматривать в качестве модулей. В Scala не возникает потребности в «измельчении» объектов, равно как и в использовании неких конструкций, подобных модулям, для их укрупнения. Масштабируемость Scala в немалой степени определяется тем обстоятельством, что одни и те же конструкции используются как для мелких, так и для крупных структур.

Например, один из модулей будет сделан на основе одного из имитируемых компонентов уровня предметной области, который является объектом, представляющим собой базу данных. На уровне приложения объект «обозреватель базы данных» будет считаться модулем. В базе данных будут храниться все рецепты, собранные каким-либо сотрудником. Обозреватель поможет в поиске в базе данных и в просмотре ее записей, к примеру для нахождения всех рецептов, включающих имеющийся у вас ингредиент.

Сначала нужно смоделировать продукты и рецепты. Чтобы ничего не усложнять, продукт, как показано в листинге 29.1, имеет только название. А у рецепта, как показано в листинге 29.2, будут название, список ингредиентов и ряд инструкций.

Листинг 29.1. Простой класс Food для работы с продуктами

```
package org.stairwaybook.recipe
abstract class Food(val name: String) {
```

```
  override def toString = name
}
```

Листинг 29.2. Простой класс Recipe для работы с рецептами

```
package org.stairwaybook.recipe

class Recipe(
  val name: String,
  val ingredients: List[Food],
  val instructions: String
) {
  override def toString = name
}
```

Классы `Food` и `Recipe`, показанные в листингах 29.1 и 29.2, представляют *элементы предметной области*, которые будут храниться в базе данных[136]. В листинге 29.3 показаны отдельные экземпляры этих классов, которые могут применяться при написании тестов.

Листинг 29.3. Примеры экземпляров Food и Recipe для использования в тестах

```
package org.stairwaybook.recipe

object Apple extends Food("Apple")
object Orange extends Food("Orange")
object Cream extends Food("Cream")
object Sugar extends Food("Sugar")

object FruitSalad extends Recipe(
```

```
    "fruit salad",
    List(Apple, Orange, Cream, Sugar),
    "Stir it all together."
  )
```

Для модулей в Scala используются объекты, поэтому сборку вашей программы можно начать с создания двух синглтон-объектов, которые в ходе тестирования послужат имитаторами реализации модулей базы данных и обозревателя. Поскольку это будут имитаторы, модуль базы данных основан на простом хранящемся в памяти списке. Реализации этих объектов показаны в листинге 29.4.

Листинг 29.4. Модули-имитаторы базы данных и обозревателя

```
package org.stairwaybook.recipe

object SimpleDatabase {
  def allFoods = List(Apple, Orange, Cream, Sugar)

  def foodNamed(name: String): Option[Food] =
    allFoods.find(_.name == name)

  def allRecipes: List[Recipe] = List(FruitSalad)
}

object SimpleBrowser {
  def recipesUsing(food: Food) =
    SimpleDatabase.allRecipes.filter(recipe =>
      recipe.ingredients.contains(food))
}
```

Использовать эти базу данных и обозреватель можно

следующим образом:

```
scala> val apple = SimpleDatabase.foodNamed("Apple").get
apple: Food = Apple

scala> SimpleBrowser.recipesUsing(apple)
res0: List[Recipe] = List(fruit salad)
```

Чтобы стало еще интереснее, предположим, что в базе данных продукты отсортированы по категориям. Для реализации этого свойства можно, как показано в листинге 29.5, добавить класс `FoodCategory` и внести в базу данных перечень всех категорий. Обратите внимание на то, что в этом примере ключевое слово `private`, столь полезное для реализации классов, пригодилось и для реализации модулей. Элементы, обозначенные как закрытые (`private`), являются частью реализации модуля, в силу чего в них, в частности, будут просто вносить изменения, не оказывающие никакого влияния на все остальные модули.

На данном этапе могут быть добавлены и многие другие объекты, но главное сейчас — ухватить саму идею. Программы можно разделить на синглтон-объекты, о которых можно думать, что это модули. В этом нет ничего особенного, но такой подход к делу окажется весьма полезен при обдумывании абстракции, рассматриваемой далее.

Листинг 29.5. Модули базы данных и обозревателя с добавлением категорий

```
package org.stairwaybook.recipe

object SimpleDatabase {
  def allFoods = List(Apple, Orange, Cream, Sugar)
```

```
  def foodNamed(name: String): Option[Food] =
    allFoods.find(_.name == name)

  def allRecipes: List[Recipe] = List(FruitSalad)
   case class FoodCategory(name: String, foods:
List[Food])

  private var categories = List(
    FoodCategory("fruits", List(Apple, Orange)),
    FoodCategory("misc", List(Cream, Sugar)))

  def allCategories = categories
}

object SimpleBrowser {
  def recipesUsing(food: Food) =
    SimpleDatabase.allRecipes.filter(recipe =>
      recipe.ingredients.contains(food))

               def       displayCategory(category:
SimpleDatabase.FoodCategory) = {
    println(category)
  }
}
```

29.3. Абстракция

С помощью приведенных выше примеров ваше приложение удалось разбить на отдельные модули базы данных и обозревателя, однако конструкция все еще не обладает достаточной модульностью. Дело в том, что в ней имеется, по сути, жесткая

ссылка из модуля обозревателя на модуль базы данных

```
SimpleDatabase.allRecipes.filter(recipe => ...
```

Поскольку в модуле `SimpleBrowser` упоминается по имени модуль `SimpleDatabase`, вы не сможете подключить другую реализацию модуля базы данных без внесения изменений и перекомпиляции модуля обозревателя. Кроме того, хотя из `SimpleDatabase` на модуль `SimpleBrowser` жесткой ссылки нет[137], не вполне понятно, как, к примеру, позволить уровню пользовательского интерфейса быть перенастроенным на использование другой реализации модуля обозревателя.

Но при повышении подключаемости модулей важно избегать повторяемости кода, поскольку различные реализации одного и того же модуля будут, скорее всего, совместно использовать большой объем кода. Предположим, к примеру, что для поддержки баз данных с рецептами требуется одна и та же кодовая база, но при этом вам нужна возможность создания для каждой из этих баз данных отдельного обозревателя. Для любого из этих экземпляров вам захочется повторно использовать код обозревателя, поскольку обозреватели отличаются друг от друга лишь той базой данных, к которой они обращаются. Кроме реализации, касающейся базы данных, весь остальной код может быть использован повторно без малейших изменений. Как построить программу таким образом, чтобы свести продублированность кода к минимуму? Как сделать код перенастраиваемым, чтобы его можно было настроить под использование любой реализации базы данных?

Ответ известен: если модуль является объектом, то шаблоном для модуля выступает класс. Точно так же как класс может содержать описание общих частей всех своих экземпляров, он может описывать части модуля, являющиеся общими для всех его возможных конфигураций.

Поэтому определение обозревателя становится классом, а не объектом, а указание на используемую базу данных является, как

показано в листинге 29.6, абстрактным элементом класса.

Листинг 29.6. Класс Browser с абстрактной val-переменной database

```
abstract class Browser {
  val database: Database

  def recipesUsing(food: Food) =
    database.allRecipes.filter(recipe =>
      recipe.ingredients.contains(food))

                def        displayCategory(category:
database.FoodCategory) = {
    println(category)
  }
}
```

База данных также становится классом, включающим по максимуму все общее, что есть у всех баз данных, и объявляющим недостающие части, которые должна определять база данных. В этом случае во всех модулях баз данных должны определяться методы для `allFoods`, `allRecipes` и `allCategories`, но, поскольку в них может использоваться произвольное определение, в классе `Database` методы должны оставаться абстрактными. В отличие от них, метод `foodNamed`, как показано в листинге 29.7, может быть определен в абстрактном классе `Database`.

Листинг 29.7. Класс Database с абстрактными методами

```
abstract class Database {
  def allFoods: List[Food]
  def allRecipes: List[Recipe]
```

```
  def foodNamed(name: String) =
    allFoods.find(f => f.name == name)

  case class FoodCategory(name: String, foods:
List[Food])
  def allCategories: List[FoodCategory]
}
```

Объект `SimpleDatabase` нужно обновить, чтобы он, как показано в листинге 29.8, стал наследником абстрактного класса `Database`.

Листинг 29.8. Объект SimpleDatabase как подкласс класса Database

```
object SimpleDatabase extends Database {
  def allFoods = List(Apple, Orange, Cream, Sugar)

  def allRecipes: List[Recipe] = List(FruitSalad)

  private var categories = List(
    FoodCategory("fruits", List(Apple, Orange)),
    FoodCategory("misc", List(Cream, Sugar)))

  def allCategories = categories
}
```

Затем, как показано в листинге 29.9, путем создания экземпляра класса `Browser` и указания, какую именно базу данных использовать, создается конкретный модуль-обозреватель.

Листинг 29.9. Объект SimpleBrowser как подкласс класса Browser

```
object SimpleBrowser extends Browser {
  val database = SimpleDatabase
}
```

Эти более пригодные к подключению модули можно использовать, как и прежде:

```
scala> val apple = SimpleDatabase.foodNamed("Apple").get
apple: Food = Apple

scala> SimpleBrowser.recipesUsing(apple)
res1: List[Recipe] = List(fruit salad)
```

Вот теперь можно приступить к созданию имитатора второй базы данных и воспользоваться для него, как показано в листинге 29.10, тем же классом обозревателей.

Листинг 29.10. Обучающая база данных и обозреватель

```
object StudentDatabase extends Database {
  object FrozenFood extends Food("FrozenFood")

  object HeatItUp extends Recipe(
    "heat it up",
    List(FrozenFood),
    "Microwave the 'food' for 10 minutes.")

  def allFoods = List(FrozenFood)
  def allRecipes = List(HeatItUp)
  def allCategories = List(
    FoodCategory("edible", List(FrozenFood)))
}
```

```
object StudentBrowser extends Browser {
  val database = StudentDatabase
}
```

29.4. Разбиение модулей на трейты

Зачастую модуль оказывается слишком большим, чтобы комфортно уместиться в одном файле. В таком случае, чтобы разбить модуль на несколько отдельных файлов, можно воспользоваться трейтами. Предположим, к примеру, что вам нужно переместить код категоризации за пределы основного файла `Database`, поместив его в собственный файл. Для этого кода, как показано в листинге 29.11, можно создать трейт.

Листинг 29.11. Трейт для категорий продуктов

```
trait FoodCategories {
    case class FoodCategory(name: String, foods:
List[Food])
  def allCategories: List[FoodCategory]
}
```

Теперь, как показано в листинге 29.12, вместо того чтобы определять `FoodCategory` и `allCategories` в классе `Database`, этот класс можно подмешать в трейт `FoodCategories`.

Листинг 29.12. Класс Database, подмешиваемый в трейт FoodCategories

```
abstract class Database extends FoodCategories {
  def allFoods: List[Food]
  def allRecipes: List[Recipe]
```

```
  def foodNamed(name: String) =
    allFoods.find(f => f.name == name)
}
```

Можно постараться и разбить SimpleDatabase на два трейта — один для продуктов, а другой для рецептов. Это позволит определить SimpleDatabase так, как показано в листинге 29.13.

Листинг 29.13. Объект SimpleDatabase, состоящий исключительно из примесей

```
object SimpleDatabase extends Database
with SimpleFoods with SimpleRecipes
```

Трейт SimpleFoods может выглядеть так, как показано в листинге 29.14.

Листинг 29.14. Трейт SimpleFoods

```
trait SimpleFoods {
  object Pear extends Food("Pear")
  def allFoods = List(Apple, Pear)
  def allCategories = Nil
}
```

Пока вроде бы все получается, но, к сожалению, при попытке определения трейта для SimpleRecipes следующим образом возникнет проблема:

```
trait SimpleRecipes { // Не пройдет компиляцию
  object FruitSalad extends Recipe(
    "fruit salad",
    List(Apple, Pear), // Опа!
```

```
    "Mix it all together."
  )
  def allRecipes = List(FruitSalad)
}
```

Дело в том, что `Pear` находится в другом трейте, а не в том, который его использует, следовательно, пребывает вне области видимости. У компилятора нет никаких догадок насчет того, что `SimpleRecipes` когда-либо смешивается с `SimpleFoods`.

Но компилятору можно сообщить об этом. Именно для такой ситуации в Scala имеется *self-тип*. С технической точки зрения self-тип является предполагаемым типом для `this`, когда `this` упоминается в классе. Прагматически self-тип указывает требования к любому конкретному классу, в который подмешан данный трейт. Если имеется трейт, используемый когда-либо при смешивании с другим трейтом или трейтами, можно указать, что эти другие трейты должны предполагаться. В нашем варианте, как показано в листинге 29.15, достаточно указать self-тип `SimpleFoods`.

Листинг 29.15. Трейт SimpleRecipes с self-типом

```
trait SimpleRecipes {
  this: SimpleFoods =>

  object FruitSalad extends Recipe(
    "fruit salad",
     List(Apple, Pear), // Теперь Pear находится в
области видимости
    "Mix it all together."
  )
  def allRecipes = List(FruitSalad)
}
```

Теперь благодаря новому self-типу `Pear` находится в области видимости. Подразумевается, что ссылка на `Pear` должна выглядеть как `this.Pear`. Эта ссылка безопасна, поскольку любой конкретный класс, подмешанный в `SimpleRecipes`, должен также быть подтипом `SimpleFoods`, следовательно, `Pear` будет элементом подтипа. Абстрактные подклассы и трейты не должны придерживаться этого ограничения, но, поскольку создать их экземпляры с помощью ключевого слова `new` невозможно, риск того, что ссылка `this.Pear` даст сбой, отсутствует.

29.5. Компоновка во время выполнения

Модули Scala могут быть скомпонованы во время выполнения программы, и в зависимости от вычислений в ходе выполнения программы можно решить, какие именно модули должны быть скомпонованы. Например, в листинге 29.16 показана небольшая программа, выбирающая базу данных, а затем выводящая на стандартное устройство все имеющиеся в ней рецепты, связанные с яблоками.

Листинг 29.16. Приложение, динамически выбирающее реализацию модуля

```
object GotApples {
  def main(args: Array[String]) = {
    val db: Database =
      if(args(0) == "student")
        StudentDatabase
      else
        SimpleDatabase

    object browser extends Browser {
```

```
      val database = db
    }

                              val       apple       =
SimpleDatabase.foodNamed("Apple").get

    for(recipe <- browser.recipesUsing(apple))
      println(recipe)
  }
}
```

Теперь, если воспользоваться простой базой данных, найдется рецепт фруктового салата. Если же прибегнуть к обучающей базе данных, то для яблок не найдется вообще никаких рецептов:

```
$ scala GotApples simple
fruit salad
$ scala GotApples student
$
```

Конфигурирование с помощью кода Scala

Поскольку объект `GotApples`, показанный в листинге 29.16, содержит жесткие ссылки как на `StudentDatabase`, так и на `SimpleDatabase`, может возникнуть вопрос: не отступаем ли мы от решения проблемы жестких ссылок в исходных примерах данной главы? Разница в том, что жесткие ссылки локализованы в одном файле, который может быть заменен.

Каждое модульное приложение нуждается в каком-либо способе указания на ту действующую реализацию модуля, которая используется в конкретной ситуации. Это действие

конфигурирования приложения будет по определению затрагивать имена конкретных реализаций модуля.

Например, в приложении Spring конфигурирование путем указания имен реализаций осуществляется во внешнем XML-файле. В Scala конфигурирование можно выполнять посредством самого кода Scala. Одно из преимуществ использования для конфигурирования исходного кода на Scala перед XML заключается в том, что процесс прогона вашего конфигурационного файла через компилятор Scala вскроет в нем все опечатки еще до того, как вы начнете его применять на практике.

29.6. Отслеживание экземпляров модулей

Несмотря на использование одного и того же кода, различные модули обозревателей и баз данных, созданные в предыдущем разделе, в действительности являются отдельными модулями. Это означает, что у каждого модуля есть собственное содержимое, включая любые вложенные классы. К примеру, `FoodCategory` в `SimpleDatabase` — это класс, который отличается от `FoodCategory` в `StudentDatabase`!

```
scala> val category = StudentDatabase.allCategories.head
category: StudentDatabase.FoodCategory = FoodCategory(edible,List(FrozenFood))

scala> SimpleBrowser.displayCategory(category)
<console>:21: error: type mismatch;
found : StudentDatabase.FoodCategory
```

```
required: SimpleBrowser.database.FoodCategory
                SimpleBrowser.displayCategory(categor
                                                    ^
```

Но если вы предпочитаете, чтобы все `FoodCategory` представляли собой одно и то же, то добиться этого можно, переместив определение `FoodCategory` за пределы любого класса или трейта.

Выбор за вами, но в том виде, в котором это уже написано, каждый класс `Database` получает собственный уникальный класс `FoodCategory`.

Поскольку два класса `FoodCategory`, показанные в данном примере, действительно разные, компилятор вправе пожаловаться. Но иногда можно столкнуться со случаем, когда два типа одинаковы, но компилятор не в состоянии проверить это. Вы увидите: компилятор жалуется на то, что эти два типа разные, при том что вы как программист знаете, каковы они на самом деле.

В таких случаях зачастую удается решить задачу, используя *синглтон-типы*. К примеру, в программе `GotApples` система проверки соответствия типов не знает, что `db` и `browser.database` — это одно и то же. Попытка передавать категории между двумя объектами повлечет за собой возникновение ошибок несоответствия типов:

```
object GotApples {
  // одинаковые определения...

  for (category <- db.allCategories)
    browser.displayCategory(category)

  // ...
}
```

```
GotApples2.scala:14: error: type mismatch;
found : db.FoodCategory
required: browser.database.FoodCategory
        browser.displayCategory(category)
                                ^
one error found
```

Для предотвращения ошибки нужно проинформировать систему проверки соответствия типов о том, что они являются одним и тем же объектом. Сделать это можно, изменив определение `browser.database`, как показано в листинге 29.17.

Листинг 29.17. Использование синглтон-типа

```
object browser extends Browser {
  val database: db.type = db
}
```

Это определение такое же, как и прежде, за исключением того, что у `database` имеется забавно выглядящий тип `db.type`. Часть `.type` в конце означает, что это синглтон-тип, который имеет совершенно особую природу и содержит только один объект, в данном случае именно тот, на который ссылается `db`. Обычно такие типы слишком специфичны, чтобы быть полезными, именно поэтому компилятор не желает вставлять их автоматически. Но в таком случае синглтон-тип позволяет компилятору узнать, что `db` и `browser.database` — это один и тот же объект, то есть дает достаточно информации, чтобы избежать появления ошибки несоответствия типов.

Резюме

В этой главе рассматривалось использование объектов Scala в качестве модулей. Данный подход раскрывает перед вами

разнообразные способы создания не только простых статических модулей, но и абстрактных, перенастраиваемых модулей. В действительности существует гораздо больше технологий создания абстракций, чем было показано, поскольку все, что работает с классами, работает и с классом, используемым для реализации модуля. Как всегда, в какой мере воспользоваться этим широким разнообразием возможностей — дело вкуса.

Модули являются составной частью программирования с прицелом на большое, и поэтому с ними сложно экспериментировать. Чтобы реально изменить ситуацию, понадобится какая-нибудь крупная программа. И все же, прочитав главу, вы узнали, каким свойствам Scala следует уделить внимание при желании перейти к программированию в модульном стиле. Теперь вы можете поразмышлять о применении этих технологий при написании собственных крупных программ и распознать показанные здесь программные шаблоны, когда они встретятся в чьем-нибудь программном коде.

133 Эта терминология была введена в книге DeRemer, Frank and Hans Kron. Programming-in-the large versus programming-in-the-small // In Proceedings of the international conference on Reliable software. — New York: ACM, 1975. — P. 114–121. http://doi.acm.org/10.1145/8 00027.8 08431.

134 Fowler M. Inversion of Control Containers and the Dependency Injection pattern. January 2004 // http://martinfowler.com/articles/injection.html (accessed August 6, 2008).

135 Названия этих уровней соответствуют терминологии, принятой в публикации Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. — Addison-Wesley Professional, 2003.

136 Здесь, чтобы не засорять примеры обилием подробностей из реального мира, эти классы элементов предметной области представлены в упрощенном виде. Но превращение этих классов, к примеру, в элементы предметной области, которые могут сохраняться с помощью Hibernate или Java Persistence Architecture, потребует внесения всего лишь нескольких изменений, таких как добавление закрытого поля id типа Long и конструктора, не использующего аргументов, помещение в отношении полей аннотации scala.reflect.BeanProperty, указывающей на соответствующее отображение посредством аннотаций, или же отдельного XML-файла и т. д.

137 И в этом нет ничего плохого, поскольку каждый из этих архитектурных уровней должен зависеть только от того уровня, который находится под ним.

30. Равенство объектов

Сравнение двух значений встречается в программировании практически повсеместно. Выполнить эту операцию порой не так просто, как кажется на первый взгляд. В этой главе подробно рассматривается равенство объектов и дается ряд рекомендаций по разработке собственных тестов на равенство.

30.1. Понятие равенства в Scala

Как упоминалось в разделе 11.2, способы определения равенства в Scala и в Java отличаются друг от друга. В Java их два: с использованием оператора `==`, применяемого обычно для определения равенства типов значений и идентичности объектов ссылочных типов, и с использованием метода `equals` — для выявления классического равенства (определяемого пользователем) ссылочных типов. Это положение не лишено проблем, поскольку более натуральный символ `==` не всегда соответствует естественному понятию равенства. При программировании на Java новички часто спотыкаются на сравнении объектов с применением оператора `==`, в то время как их следует проверять на равенство с помощью метода `equals`. Например, сравнение двух строк, `x` и `y`, с использованием выражения `x == y` может в Java выдать значение `false`, даже если у `x` и `y` имеются одинаковые символы, расположенные в одном и том же порядке.

В Scala также имеется метод определения равенства, обозначающий идентичность объектов, но он используется нечасто. Эта разновидность определения равенства, записываемая в виде `x eq y`, вычисляется в `true`, если `x` и `y` ссылаются на один и тот же объект. Знак равенства `==` зарезервирован в Scala для обозначения естественного равенства каждого типа. Для типов

значений == обозначает, как и в Java, сравнение значений. Для ссылочных типов == является в Scala аналогом метода equals. Поведение метода == для новых типов можно переопределить путем переопределения метода equals, который всегда наследуется из класса Any. Унаследованный метод equals, работающий до переопределения, выявляет, как и в Java, идентичность объектов. Следовательно, equals (а с ним заодно и ==) изначально является аналогом eq, но вы можете изменить его поведение переопределением метода equals в определяемых вами классах. Переопределить == напрямую невозможно, поскольку он в классе Any определен как терминальный метод. То есть в Scala метод == рассматривается так, будто он был определен в классе Any следующим образом:

```
final def == (that: Any): Boolean =
   if (null eq this) {null eq that} else {this
equals that}
```

30.2. Написание метода равенства

Как должен быть определен метод equals? Как ни удивительно, но оказывается, что правильно написать метод равенства в объектно-ориентированных языках не так-то просто. После изучения большого объема Java-кода авторы статьи, вышедшей в 2007 году, пришли к выводу, что почти все реализации методов equals не лишены недостатков[138]. Проблема в том, что равенство лежит в основе многих других вещей. Скажем, ошибочно определенный метод равенства для типа C может означать, что вы не сможете гарантированно поместить объект типа C в коллекцию.

Например, у вас имеются два относящихся к типу C элемента, elem1 и elem2, равных друг другу (то есть выражение elem1 equals elem2 выдает true). И тем не менее, при часто встречающихся ошибочных реализациях метода equals можно все

же столкнуться со следующим поведением программы:

```
var hashSet: Set[C] = new collection.immutable.HashSet
hashSet += elem1
hashSet contains elem2 // возвращает false!
```

Неверное поведение при переопределении метода `equals` может быть вызвано четырьмя основными просчетами[139].

Определение `equals` с неверной сигнатурой.

27. Внесение изменений в `equals` без внесения соответствующих изменений в `hashCode`.

28. Определение `equals` с применением изменяемых полей.

29. Невозможность определения `equals` в качестве отношения эквивалентности.

Далее нам предстоит рассмотреть эти четыре возможных просчета более подробно.

Просчет 1: определение equals с неверной сигнатурой

Рассмотрим добавление метода `equality` к следующему классу простых точек:

```
class Point(val x: Int, val y: Int) { ... }
```

Казалось бы, все вполне очевидно, но для определения можно выбрать следующий неверный путь:

```
// Абсолютно ошибочное определение equals
def equals(other: Point): Boolean =
```

```
    this.x == other.x && this.y == other.y
```

Что неправильно в этом методе? На первый взгляд, все работает довольно успешно:

```
scala> val p1, p2 = new Point(1, 2)
p1: Point = Point@37d7d90f
p2: Point = Point@3beb846d

scala> val q = new Point(2, 3)
q: Point = Point@e0cf182

scala> p1 equals p2
res0: Boolean = true

scala> p1 equals q
res1: Boolean = false
```

Но неприятности станут проявляться, как только начнется помещение точек в коллекцию:

```
scala> import scala.collection.mutable
import scala.collection.mutable

scala> val coll = mutable.HashSet(p1)
coll: scala.collection.mutable.HashSet[Point] =
Set(Point@37d7d90f)

scala> coll contains p2
res2: Boolean = false
```

Чем объяснить, что `coll` не содержит `p2`, тем более при том, что объект `p1` был добавлен в коллекцию, а `p1` и `p2` являются равными объектами? Причина становится понятной при

следующем взаимодействии, где точный тип одной из сравниваемых точек маскируется. Определите в качестве псевдонима p2 переменную p2a, но вместо типа Point укажите для нее тип Any:

```
scala> val p2a: Any = p2
p2a: Any = Point@3beb846d
```

Теперь при повторе первого сравнения, но уже с псевдонимом p2a, а не p2, будет получен следующий результат:

```
scala> p1 equals p2a
res3: Boolean = false
```

Что пошло не так? Используемая прежде версия equals не переопределила стандартный метод equals, поскольку у нее другой тип. А вот каким является тип метода equals, определенного в корневом классе Any[140]:

```
def equals(other: Any): Boolean
```

Поскольку метод equals в Point получает в качестве аргумента не Any, а Point, метод equals в Any не переопределяется. Вместо этого просто перегружается альтернативный вариант. Теперь перегрузка в Scala и Java разрешается за счет аргумента статического типа, а не типа, определяемого в ходе выполнения программы. А при условии, что статическим типом аргумента является Point, вызывается метод equals, который определен в Point. Но так как статический аргумент относится к типу Any, вместо этого вызывается метод equals, который определен в Any. Этот метод не был переопределен, поэтому он по-прежнему реализован на основе определения идентичности объектов.

Именно поэтому сравнение p1 equals p2a выдает значение false, хотя у точек p1 и p2a одинаковые значения x и y. Также это становится причиной того, что метод contains в HashSet

возвращает `false`. Так как этот метод работает с универсальными наборами, он вызывает универсальный метод `equals`, который определен в классе `Object`, а не перегружает его вариант, который определен в `Point`. А вот как выглядит более подходящий метод `equals`:

```
// Уже лучше, но еще не идеально
override def equals(other: Any) = other match {
   case that: Point => this.x == that.x && this.y
== that.y
  case _ => false
}
```

Теперь у `equals` правильный тип. Метод получает в качестве параметра значение типа `Any` и выдает результат типа `Boolean`. В реализации этого метода используется поиск по шаблону. Сначала в нем проверятся принадлежность других объектов к тому же типу `Point`.

Если она подтвердится, сравниваются координаты двух точек и возвращается результат. В противном случае результатом становится значение `false`.

В данном случае можно допустить просчет, определив `==` с неверной сигнатурой. Обычно, если попытаться переопределить `==` с правильной сигнатурой, получающей аргумент типа `Any`, компилятор выдаст ошибку, поскольку это будет рассматриваться как попытка перезаписи терминального метода типа `Any`.

Новички в Scala порой допускают сразу две ошибки: пытаются переопределить `==` и дают этому методу неверную сигнатуру, например:

```
def ==(other: Point): Boolean = // Не делайте
этого!
```

В данном случае определенный пользователем метод `==`

рассматривается как перегружаемый вариант метода с таким же именем в классе Any и программа проходит компиляцию. Но поведение у программы будет таким же сомнительным, как при определении equals с неверной сигнатурой.

Просчет 2: изменение equals без соответствующего изменения hashCode

Продолжим рассматривать пример из просчета 1. Если повторить сравнение p1 и p2a с самым последним определением Point, будет получен, как и ожидалось, результат true. Но если повторить тест HashSet.contains, то, наверное, опять будет получен результат false:

```
scala> val p1, p2 = new Point(1, 2)
p1: Point = Point@122c1533
p2: Point = Point@c23d097

scala> collection.mutable.HashSet(p1) contains p2
res4: Boolean = false
```

Но такой исход вероятен не на все 100 %. В ходе эксперимента можно также получить true. Если результат именно такой, можно поэкспериментировать с какими-нибудь другими точками с координатами 1 и 2. Со временем найдется одна, не присутствующая в наборе. Дело в том, что в Point метод equals переопределен без сопутствующего ему переопределения метода hashCode.

Учтите, что коллекция в данном примере относится к типу HashSet. Следовательно, элементы коллекции помещаются в участки оперативной хеш-памяти, определяемые их хеш-кодом. Тест на принадлежность сначала определяет хеш-ведро, в которое следует заглянуть, а затем сравнивает заданные элементы со всеми элементами в этом участке памяти. Теперь последняя версия класса Point обновила equals, но в то же самое время не

обновила hashCode. Следовательно, hashCode остался таким же, каким он был в своей версии в классе AnyRef, — выполняющим некоторые преобразования адреса размещаемого объекта.

Хеш-коды p1 и p2 практически всегда разные, даже при том что поля обеих точек имеют одинаковые значения. А разные хеш-коды с высокой степенью вероятности означают разные участки хеш-памяти в наборе. Тест на принадлежность станет искать соответствующие элементы в участке хеш-памяти с хеш-кодом, соответствующим хеш-коду p2. В большинстве случаев точка p1 окажется в другом участке, следовательно, она никогда не будет найдена. Точки p1 и p2 могут случайно оказаться и в одном и том же участке хеш-памяти. Тогда тест возвратит значение true. Проблема в том, что эта полезная реализация Point нарушила соглашение в отношении hashCode, определенное для класса Any[141]: если два объекта равны с точки зрения метода equals, то вызов метода hashCode в отношении каждого из двух объектов должен выдавать одинаковый результат.

Фактически в Java хорошо известно, что hashCode и equals должны всегда переопределяться вместе. Более того, hashCode может зависеть только от тех полей, от которых зависит equals. Для класса Point подходящее определение hashCode могло бы быть выражено следующим кодом:

```
class Point(val x: Int, val y: Int) {
  override def hashCode = (x, y).##
  override def equals(other: Any) = other match {
    case that: Point => this.x == that.x && this.y
== that.y
    case _ => false
  }
}
```

Это всего лишь одна из множества возможных реализаций hashCode. Следует напомнить, что метод ## является

сокращенной формой записи метода для вычисления хеш-кодов, работающего с элементарными значениями, ссылочными типами и null. При вызове в отношении коллекции или кортежа он вычисляет смешанный хеш, реагирующий на хеш-коды всех элементов в коллекции. Дополнительные указания по написанию кода метода hashCode будут даны чуть позже.

Добавление hashCode решает проблемы выявления равенства при определении таких классов, как Point, но есть и другие тревожные моменты, которые нужно отслеживать.

Просчет 3: определение equals с использованием изменяемых полей

Рассмотрим следующую небольшую вариацию класса Point:

```
class Point(var x: Int, var y: Int) { // Проблемное место
  override def hashCode = (x, y).##
  override def equals(other: Any) = other match {
    case that: Point => this.x == that.x && this.y == that.y
    case _ => false
  }
}
```

Единственное различие заключается в том, что теперь поля x и y относятся к var-, а не к val-переменным. Сейчас методы equals и hashCode определены относительно этих изменяемых полей, следовательно, их результаты изменяются при изменении полей. Если поместить точки в коллекции, это может привести к странным последствиям:

```
scala> val p = new Point(1, 2)
p: Point = Point@5428bd62
```

```
scala> val coll = collection.mutable.HashSet(p)
coll: scala.collection.mutable.HashSet[Point] =
Set(Point@5428bd62)

scala> coll contains p
res5: Boolean = true
```

Теперь, если изменить поле в точке `p`, будет ли коллекция по-прежнему содержать эту точку? Давайте посмотрим:

```
scala> p.x += 1

scala> coll contains p
res7: Boolean = false
```

Выглядит странно. Куда же делась `p`? Еще более странные результаты будут получены, если проверить, содержит ли `p` итератор набора:

```
scala> coll.iterator contains p
res8: Boolean = true
```

Получается, есть набор, не содержащий `p`, но тем не менее `p` находится среди элементов набора! Получилось, что после изменения поля `x` точка `p` оказалась в другом участке хеш-памяти, не в том, где набор `coll`. То есть исходный участок хеш-памяти больше не соответствует новому значению хеш-кода точки. Иначе говоря, точка `p` исчезла из поля зрения набора `coll`, хотя она еще принадлежала его элементам.

Из этого примера нужно извлечь урок: когда `equals` и `hashCode` зависят от изменяемого состояния, возникают проблемы для потенциальных пользователей. Если поместить такие объекты в коллекцию, придется проявлять осмотрительность и никогда не изменять зависимое состояние, а сделать это не так-

то просто. Если возникнет потребность в сравнении, которое принимает в расчет текущее состояние объекта, то его следует назвать как-то иначе, но никак не `equals`.

Относительно последнего определения `Point`: было бы предпочтительнее избавиться от переопределения `hashCode` и назвать метод сравнения `equalContents` или каким-нибудь другим, отличающимся от `equals` именем. Тогда `Point` унаследует исходную реализацию `equals` и `hashCode`, а `p` будет размещаться в `coll` даже после изменения его поля `x`.

Просчет 4: невозможность определения equals в качестве отношения эквивалентности

В соглашении по методу `equals` в `scala.Any` указывается, что в `equals` должны быть реализованы отношения эквивалентности для объектов, не имеющих значения `null`[142], выражающиеся в совокупности следующих свойств:

- **рефлексивности** — для любого отличного от `null` значения `x` выражение `x.equals(x)` должно возвращать `true`;
- **симметричности** — для любых отличных от `null` значений `x` и `y` выражение `x.equals(y)` должно возвращать `true` исключительно в тех случаях, когда выражение `y.equals(x)` возвращает `true`;
- **транзитивности** — если для любых отличных от `null` значений `x`, `y` и `z` выражение `x.equals(y)` возвращает `true` и `y.equals(z)` возвращает `true`, то и `x.equals(z)` должно возвращать `true`;
- **постоянства** — для любых отличных от `null` значений `x` и `y` множественные вызовы `x.equals(y)` должны неизменно возвращать `true` или же неизменно возвращать `false` при

условии, что информация в сравнениях объектов на эквивалентность не изменена. Для любого отличного от null значения x выражение x.equals(null) должно возвращать false.

Определение equals, разработанное для класса Point, на данный момент удовлетворяет соглашению по equals. Но все усложняется, как только дело касается подклассов. Предположим, что существует подкласс ColoredPoint класса Point, который добавляет поле color, имеющее тип Color. Также предположим, что Color определен как перечисление, представленное в разделе 20.9:

```
object Color extends Enumeration {
   val Red, Orange, Yellow, Green, Blue, Indigo,
Violet = Value
}
```

В ColoredPoint метод equals переопределяется, чтобы учитывалось новое поле color:

```
class ColoredPoint(x: Int, y: Int, val color:
Color.Value)
    extends Point(x, y) {   // Проблема: утрачена
симметричность equals
  override def equals(other: Any) = other match {
    case that: ColoredPoint =>
                this.color   ==   that.color   &&
super.equals(that)
    case _ => false
}
}
```

Именно такой код, скорее всего, напишут многие

программисты. Обратите внимание, что классу ColoredPoint не нужно переопределять метод hashCode. Поскольку новое определение equals в ColoredPoint строже, чем переделанное определение в Point (то есть он определяет эквивалентность меньшего количества пар объектов), соглашение по поводу hashCode остается ненарушенным. Если две цветные точки эквивалентны, у них должны быть одинаковые координаты, следовательно, гарантируется равенство и их хеш-кодов.

Если ограничиться рассмотрением лишь класса ColoredPoint как такового, то в имеющемся в нем определении equals вроде бы все в порядке. Но соглашение по equals нарушено, поскольку смешиваются простые и цветные точки. Рассмотрим следующий диалог с интерпретатором:

```
scala> val p = new Point(1, 2)
p: Point = Point@5428bd62

scala> val cp = new ColoredPoint(1, 2, Color.Red)
cp: ColoredPoint = ColoredPoint@5428bd62

scala> p equals cp
res9: Boolean = true

scala> cp equals p
res10: Boolean = false
```

Сравнение p equals cp вызывает метод equals, имеющийся в объекте p, который определен в классе Point. Этот метод берет в расчет только координаты двух точек. Следовательно, сравнение выдает true. В то же время сравнение cp equals p вызывает метод equals, имеющийся в объекте cp, который определен в классе ColoredPoint. Этот метод возвращает false, поскольку p не относится к типу ColoredPoint. Следовательно, отношение,

определенное в методе equals, несимметрично.

Утрата симметричности может привести к неожиданным последствиям для коллекций. Рассмотрим следующий пример:

```
scala>         collection.mutable.HashSet[Point](p)
contains cp
res11: Boolean = true

scala>        collection.mutable.HashSet[Point](cp)
contains p
res12: Boolean = false
```

Даже если p и cp эквивалентны, один объект проходит проверку на принадлежность к коллекции contains успешно, а второй ее не проходит.

Как изменить определение equals, чтобы добиться симметричности? Есть два способа. Отношение можно сделать либо более общим, либо более строгим. Превращение его в более общее означает, что пара объектов, x и y, считается эквивалентной, если сравнение x с y или y с x выдает true. Рассмотрим код, выполняющий этот замысел:

```
class ColoredPoint(x: Int, y: Int, val color:
Color.Value)
     extends Point(x, y) { // Проблема: нарушена
транзитивность equals
  override def equals(other: Any) = other match {
    case that: ColoredPoint =>
                (this.color   ==   that.color)   &&
super.equals(that)
    case that: Point =>
      that equals this
    case _ =>
      false
```

```
    }
  }
```

В новом определении equals в ColoredPoint на один case-вариант больше, чем в старом: если другой объект относится к Point, а не к ColoredPoint, метод передает управление методу equals класса Point. Тем самым достигается желаемый эффект симметричности equals. Теперь оба выражения, и cp equals p, и p equals cp, выдают результат true. Но соглашение относительно equals по-прежнему нарушено. Дело в том, что новое отношение больше не обладает транзитивностью!

Вот как выглядит последовательность инструкций, демонстрирующая это. Определите место и две точки разного цвета так, чтобы все они были в одной и той же позиции:

```
scala> val redp = new ColoredPoint(1, 2, Color.Red)
redp: ColoredPoint = ColoredPoint@5428bd62

scala> val bluep = new ColoredPoint(1, 2, Color.Blue)
bluep: ColoredPoint = ColoredPoint@5428bd62
```

По отдельности redp эквивалентна p, а p эквивалентна bluep:

```
scala> redp == p
res13: Boolean = true
scala> p == bluep
res14: Boolean = true
```

А вот сравнение redp и bluep выдает false:

```
scala> redp == bluep
res15: Boolean = false
```

Следовательно, условие транзитивности в соглашении по `equals` нарушено.

Похоже, если сделать отношения эквивалентности более общими, это заведет нас в тупик. Вместо этого попробуем сделать их более строгими. Один из способов повышения строгости достигается неизменным рассмотрением объектов разных классов в качестве разных объектов. Добиться этого можно за счет изменения методов `equals` в классах `Point` и `ColoredPoint`. В классе `Point` можно добавить еще одно сравнение, проверяющее, что класс на этапе выполнения программы другого `Point`-объекта такой же, как и класс данного `Point`-объекта:

```
// Технически состоятельный, но не удовлетворяющий
нас метод equals
class Point(val x: Int, val y: Int) {
  override def hashCode = (x, y).##
  override def equals(other: Any) = other match {
    case that: Point =>
      this.x == that.x && this.y == that.y &&
      this.getClass == that.getClass
    case _ => false
  }
}
```

Затем можно вернуть реализацию класса `ColoredPoint` назад — к той версии, которая ранее была забракована из-за несоблюдения требований симметричности[143]:

```
class ColoredPoint(x: Int, y: Int, val color:
Color.Value)
    extends Point(x, y) {

  override def equals(other: Any) = other match {
    case that: ColoredPoint =>
```

```
                (this.color == that.color) &&
super.equals(that)
    case _ => false
  }
}
```

Здесь экземпляр класса `Point` считается эквивалентным какому-либо другому экземпляру того же класса лишь при условии, что у объекта имеются такие же координаты и одинаковый класс на этапе выполнения, что означает возвращение методом `getClass` в отношении обоих объектов одного и того же значения. Новое определение отвечает требованиям симметричности и транзитивности, поскольку теперь каждое сравнение объектов разных классов выдает `false`. Следовательно, цветная точка никогда не будет эквивалентна простой точке. Это соглашение представляется вполне резонным, но излишняя строгость нового определения некоторыми может быть оспорена.

Рассмотрим следующий непрямой способ определения точки с координатами (1, 2):

```
scala> val pAnon = new Point(1, 1) { override val
y = 2 }
pAnon: Point = $anon$1@5428bd62
```

Будет ли объект `pAnon` эквивалентен объекту `p`? Нет, не будет, поскольку объекты `java.lang.Class`, ассоциируемые с `p` и `pAnon`, разные. Для `p` это `Point`, а для `pAnon` — безымянный подкласс `Point`. Но вполне очевидно, что `pAnon` является всего лишь еще одной точкой с координатами (1, 2). Неразумно считать ее отличной от `p`.

Похоже, мы зашли в тупик. Существует ли разумный способ переопределения эквивалентности на нескольких уровнях иерархии классов с соблюдением всех условий соглашения? Фактически такой способ существует, но для него нужен еще один

метод, переопределяемый вместе с `equals` и `hashCode`. Идея заключается в том, что как только в классе переопределяется `equals` (и `hashCode`), должно быть также прямо указано, что объекты этого класса неэквивалентны объектам какого-либо родительского класса, в котором реализован другой метод `equality`. Это достигается добавлением к каждому классу, в котором переопределен `equals`, метода `canEqual`. Этот метод имеет следующую сигнатуру:

```
def canEqual(other: Any): Boolean
```

Метод должен вернуть либо `true`, если объект `other` является экземпляром класса, в котором определен (или переопределен) метод `canEqual`, либо `false` в противном случае. Его вызывают из `equals`, чтобы убедиться, что объекты можно сравнивать в обоих направлениях. Новое (и окончательное) определение класса `Point`, придерживающееся этого принципа, показано в листинге 30.1.

Листинг 30.1. Метод equals родительского класса, вызывающий метод canEqual

```
class Point(val x: Int, val y: Int) {
  override def hashCode = (x, y).##
  override def equals(other: Any) = other match {
    case that: Point =>
      (that canEqual this) &&
      (this.x == that.x) && (this.y == that.y)
    case _ =>
     false
  }
          def     canEqual(other:     Any)     =
other.isInstanceOf[Point]
}
```

В этой версии метода equals класса Point содержится дополнительное требование: другой объект может быть эквивалентен этому объекту согласно определению в методе canEqual. В реализации canEqual в классе Point утверждается, что эквивалентными могут быть все экземпляры класса Point.

В листинге 30.2 показана соответствующая реализация ColoredPoint. Можно показать, что новые определения Point и ColoredPoint придерживаются соглашения по equals. Эквивалентность симметрична и транзитивна. Сравнение экземпляров Point с экземплярами ColoredPoint всегда выдает false. Разумеется, для любой точки p и цветной точки cp при вычислении выражения p equals cp будет возвращено значение false, поскольку при вычислении cp canEqual p будет получено значение false. Обратное сравнение, cp equals p, также приведет к возвращению false, потому что p не относится к типу ColoredPoint, следовательно, первое сопоставление с шаблоном в теле equals в классе ColoredPoint завершится неудачей.

Листинг 30.2. Метод equals подкласса, вызывающий метод canEqual

```
class ColoredPoint(x: Int, y: Int, val color:
Color.Value)
    extends Point(x, y) {

     override def hashCode = (super.hashCode,
color).##
  override def equals(other: Any) = other match {
    case that: ColoredPoint =>
      (that canEqual this) &&
            super.equals(that) && this.color ==
that.color
    case _ =>
      false
```

```
  }
  override def canEqual(other: Any) =
    other.isInstanceOf[ColoredPoint]
}
```

В то же время экземпляры различных подклассов `Point` могут быть эквивалентны при условии, что ни в одном из них метод `equality` не был переопределен. Например, после определения новых классов сравнение `p` и `pAnon` выдаст `true`. Рассмотрим несколько примеров:

```
scala> val p = new Point(1, 2)
p: Point = Point@5428bd62

scala> val cp = new ColoredPoint(1, 2, Color.Indigo)
cp: ColoredPoint = ColoredPoint@e6230d8f

scala> val pAnon = new Point(1, 1) { override val y = 2 }
pAnon: Point = $anon$1@5428bd62

scala> val coll = List(p)
coll: List[Point] = List(Point@5428bd62)

scala> coll contains p
res16: Boolean = true

scala> coll contains cp
res17: Boolean = false

scala> coll contains pAnon
res18: Boolean = true
```

В этих примерах продемонстрировано, что если реализация `equals` в родительском классе определена и в ней вызывается метод `canEqual`, то программисты, реализующие подклассы, имеют возможность решать, могут или нет их подклассы быть эквивалентны экземплярам родительского класса. Допустим, поскольку в `ColoredPoint` метод `canEqual` переопределяется, цветная точка никогда не может быть эквивалентна обычной точке. Но, поскольку в безымянных подклассах, на которые делается ссылка из `pAnon`, метод `canEqual` не переопределяется, их экземпляры могут быть эквивалентны экземпляру класса `Point`.

Подход с применением `canEqual` можно критиковать за нарушение принципа подстановки Лисков (LSP). В качестве примера нарушения LSP можно привести технологию реализации `equals` путем сравнения классов на этапе выполнения программы, которая приводит к невозможности определения подкласса, экземпляры которого могут быть эквивалентны экземплярам родительского класса144. Согласно положениям LSP должна существовать возможность использования (подстановки) экземпляра подкласса там, где требуется экземпляр родительского класса.

Но в предыдущем примере `coll contains cp` выдает `false`, несмотря на то что имеющиеся у `cp` значения `x` и `y` соответствуют таким же значениями точки в коллекции. Таким образом, это может быть похоже на нарушение LSP, поскольку вы не можете использовать `ColoredPoint` там, где ожидается `Point`. Полагаем, что это неверная интерпретация: в LSP не требуется, чтобы подкласс вел себя точно так же, как его родительский класс, просто он ведет себя в соответствии с соглашениями, принятыми в отношении своего родительского класса.

Проблема, касающаяся написания метода `equals`, который сравнивает классы на этапе выполнения программы, не в том, что нарушается LSP, а в том, что вы не получаете способа создания подкласса, чьи экземпляры могут быть эквивалентны экземплярам

родительского класса. Например, если бы в предыдущем примере мы воспользовались технологией класса на этапе выполнения программы, то выражение `coll contains pAnon` выдало бы `false`, что не соответствовало бы нашему желанию. В отличие от этого, нам хотелось, чтобы выражение `coll contains cp` выдало `false`, поскольку путем переопределения `equals` в `ColoredPoint` мы в основном добивались, чтобы фиолетовая точка с координатами (1, 2) не была эквивалентна бесцветной точке с координатами (1, 2). Следовательно, в предыдущем примере у нас была возможность передать имеющемуся в коллекции методу `contains` два различных экземпляра подкласса `Point` и получить обратно два разных ответа, оба правильных.

30.3. Определение равенства для параметризованных типов

Все методы `equals` в предыдущих примерах начинались с поиска по шаблону, которым проверялось, подходит ли тип операнда к типу класса, содержащего метод `equals`. При параметризации классов эта схема нуждается в некоторой адаптации.

Рассмотрим в качестве примера двоичные деревья. В иерархии классов, показанной в листинге 30.3, для двоичного дерева определяется абстрактный класс `Tree` с двумя альтернативными реализациями: объектом `EmptyTree` и классом `Branch`, представляющим непустые деревья. Непустое дерево составляется из элемента `elem` и из левого и правого дочерних деревьев. Тип его элемента задается параметром типа `T`.

Листинг 30.3. Иерархия для двоичных деревьев

```
trait Tree[+T] {
  def elem: T
  def left: Tree[T]
  def right: Tree[T]
```

```
}

object EmptyTree extends Tree[Nothing] {
  def elem =
                                        throw         new
NoSuchElementException("EmptyTree.elem")
  def left =
                                        throw         new
NoSuchElementException("EmptyTree.left")
  def right =
                                        throw         new
NoSuchElementException("EmptyTree.right")
}

class Branch[+T](
  val elem: T,
  val left: Tree[T],
  val right: Tree[T]
) extends Tree[T]
```

Теперь добавим к этим классам методы `equals` и `hashCode`. Для самого класса `Tree` делать ничего не нужно, поскольку предполагается, что реализация этих методов выполняется отдельно для каждой реализации абстрактного класса. Для объекта `EmptyTree` также ничего делать не нужно, поскольку исходная реализация `equals` и `hashCode`, наследуемая классом `EmptyTree` из `AnyRef`, сохраняет свою работоспособность. Помимо всего прочего, объект `EmptyTree` эквивалентен только самому себе, поэтому эквивалентность должна быть ссылочной эквивалентностью, а она унаследована из `AnyRef`.

А вот над добавлением `equals` и `hashCode` к `Branch` придется потрудиться. Значение типа `Branch` должно быть эквивалентно только другим `Branch`-значениям и только если у двух значений

имеются эквивалентные поля `elem`, `left` и `right`. Вполне естественно будет применить схему для `equals`, которая была разработана в предыдущих разделах этой главы. Тогда получится следующий код:

```
class Branch[T](
  val elem: T,
  val left: Tree[T],
  val right: Tree[T]
) extends Tree[T] {
  override def equals(other: Any) = other match {
    case that: Branch[T] => this.elem == that.elem
&&
                            this.left == that.left
&&
                                    this.right ==
that.right
    case _ => false
  }
}
```

Но при компиляции этого примера выдается предупреждение о непроверенных элементах. Повторная компиляция с ключом -unchecked выявляет следующую проблему:

```
  $ fsc -unchecked Tree.scala
Tree.scala:14: warning: non variable type-argument
T in type
pattern is unchecked since it is eliminated by
erasure
    case that: Branch[T] => this.elem == that.elem
&&
               ^
```

В предупреждении говорится, что имеется поиск по шаблону в отношении типа `Branch[T]`, а система может лишь проверить, что другой ссылкой является некая разновидность `Branch`, но не может проверить, что типом элемента дерева является `T`. Объяснение причины этого уже давалось в главе 19: типы элементов параметризированных типов уничтожаются на этапе стирания, проводимого компилятором, проверить их на этапе выполнения программы невозможно.

Так что же можно сделать? К счастью, оказывается, что проверять при сравнении двух `Branch`-объектов наличие у них одинаковых типов элементов совсем не обязательно. Вполне возможно, что два `Branch`-объекта с разными типами элементов будут эквивалентны при условии, что у них одинаковые поля. Простым примером этого может послужить `Branch`-объект, состоящий из одного `Nil`-элемента и двух пустых поддеревьев. Вероятно, любые такие `Branch`-объекты можно считать равными независимо от имеющегося у них статического типа:

```
scala> val b1 = new Branch[List[String]](Nil,
           EmptyTree, EmptyTree)
b1: Branch[List[String]] = Branch@9d5fa4f

scala> val b2 = new Branch[List[Int]](Nil,
           EmptyTree, EmptyTree)
b2: Branch[List[Int]] = Branch@56cdfc29

scala> b1 == b2
res19: Boolean = true
```

Положительный результат данного сравнения был получен с показанной ранее реализацией `equals` в `Branch`. Тем самым продемонстрировано, что тип элемента `Branch`-объекта проверен не был. Если бы он был проверен, результатом было бы значение

`false`.

Можно поспорить о том, какой из двух возможных исходов сравнения будет более естественным. В конечном счете это зависит от задуманной модели представления классов. В той модели, где параметры типа присутствуют только на этапе компиляции, вполне естественно рассматривать два `Branch`-значения, `b1` и `b2`, в качестве эквивалентных. В альтернативной модели, где параметр типа формирует часть значения объекта, так же естественно будет считать их различными. Поскольку в Scala принята модель стирания типа, параметры типа на этапе выполнения программы отсутствуют, так что `b1` и `b2` вполне естественно считать эквивалентными.

Нужно всего лишь внести маленькое изменение в формулировку метода `equals`, в результате чего предупреждение о непроверенности выдаваться не будет. Вместо обозначения элемента типа `T` воспользуйтесь буквой в нижнем регистре, например `t`:

```
case that: Branch[t] => this.elem == that.elem &&
                        this.left == that.left &&
                        this.right == that.right
```

Вспомним, что в разделе 15.2 говорилось, что параметр типа в шаблоне, начинающийся с буквы в нижнем регистре, представляет неизвестный тип. Теперь сопоставление с шаблоном вида:

```
case that: Branch[t] =>
```

будет выполняться успешно для `Branch`-значений любого типа. Параметр типа `t` представляет неизвестный тип элемента `Branch`. Его можно также заменить знаком подчеркивания, как в следующем примере, эквивалентном предыдущему примеру:

```
case that: Branch[_] =>
```

Теперь осталось лишь определить для класса Branch два других метода, hashCode и canEqual, которые сопутствуют методу equals. Вот так выглядит возможная реализация метода hashCode:

```
override def hashCode: Int = (elem, left, right).##
```

Это только одна из множества возможных реализаций. Как было показано ранее, принцип заключается в получении hashCode-значений всех полей и в их объединении. А вот так выглядит в классе Branch реализация метода canEqual:

```
def canEqual(other: Any) = other match {
  case that: Branch[_] => true
  case _ => false
}
```

В реализации метода canEqual используется сопоставление с типизированным шаблоном. Можно также сформулировать код метода с использованием isInstanceOf:

```
def canEqual(other: Any) = other.isInstanceOf[Branch[_]]
```

Если вы склонны критически оценивать все здесь увиденное, на что мы вас, собственно, и наталкиваем, у вас может появиться вопрос о предназначении знака подчеркивания в показанном ранее типе. Ведь в конце концов Branch[_] с технической точки зрения является не типом шаблона, а принадлежащим методу типом параметра. Тогда как же можно оставить некоторые его части неопределенными?

Ответ на этот вопрос будет рассматриваться в следующей главе. Branch[_] является сокращенной формой записи так называемого *подстановочного типа* (wildcard type), который, если не вдаваться в

подробности, является типом с неизвестными частями. Поэтому, даже при том что технически знак подчеркивания имеет два предназначения в сопоставлении с шаблоном и в параметре типа, используемом в коде вызова метода, по сути, предназначение одно и тоже: он позволяет пометить что-то неизвестное. Окончательная версия Branch показана в листинге 30.4.

Листинг 30.4. Параметризированные типы с equals и hashCode

```
class Branch[T](
  val elem: T,
  val left: Tree[T],
  val right: Tree[T]
) extends Tree[T] {

  override def equals(other: Any) = other match {
     case that: Branch[_] => (that canEqual this)
&&
                             this.elem == that.elem
&&
                             this.left == that.left
&&
                                      this.right ==
that.right
    case _ => false
  }
          def     canEqual(other:     Any)     =
other.isInstanceOf[Branch[_]]
    override def hashCode: Int = (elem, left,
right).##
}
```

30.4. Рецепты для equals и hashCode

В этом разделе будут рассмотрены пошаговые рецепты создания методов `equals` и `hashCode`, которых должно быть достаточно для большинства ситуаций. В качестве иллюстрации воспользуемся методами класса `Rational`, показанного в листинге 30.5.

Листинг 30.5. Класс Rational с equals и hashCode

```
class Rational(n: Int, d: Int) {

  require(d != 0)

  private val g = gcd(n.abs, d.abs)
  val numer = (if (d < 0) -n else n) / g
  val denom = d.abs / g
  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)

  override def equals(other: Any): Boolean =
    other match {

      case that: Rational =>
        (that canEqual this) &&
        numer == that.numer &&
        denom == that.denom
     case _ => false
   }
  def canEqual(other: Any): Boolean =
    other.isInstanceOf[Rational]
  override def hashCode: Int = (numer, denom).##
  override def toString =
```

```
    if (denom == 1) numer.toString else numer +
"/" + denom
}
```

Чтобы создать этот класс, удалим методы математических операторов из версии класса `Rational`, показанной в листинге 6.5. Внесем также небольшие усовершенствования в метод `toString` и изменим инициализаторы `numer` и `denom` для нормализации всех дробей, чтобы у них был положительный знаменатель (то есть чтобы преобразовать 1/–2 в –1/2).

Рецепт для equals

Рассмотрим рецепт для переопределения `equals`.

Для переопределения `equals` в нетерминальном классе создайте метод `canEqual`. В определении `equals`, унаследованном из `AnyRef` (то есть в `equals`, не переопределенном где-нибудь выше в иерархии классов), определение `canEqual` должно быть обновлено, иначе будет переопределено определение метода с тем же именем. Единственное исключение из этого требования касается терминальных классов, в которых переопределяется метод `equals`, унаследованный из `AnyRef`. Для них не могут возникать аномалии подклассов, рассмотренные в разделе 30.2, следовательно, в них не нужно определять метод `canEqual`. Типом объекта, переданного `canEqual`, должен быть `Any`:

```
def canEqual(other: Any): Boolean =
```

30. Метод `canEqual` должен выдавать `true`, если объект-аргумент является экземпляром текущего класса (то есть класса, в котором определен `canEqual`), в противном случае он должен выдавать `false`:

```
other.isInstanceOf[Rational]
```

31. В методе `equals` нужно удостовериться, что типом передаваемого объекта объявлен `Any`:

```
override def equals(other: Any): Boolean =
```

32. Тело метода `equals` следует записать в виде одного выражения сопоставления. Селектором сопоставления должен быть объект, переданный `equals`:

```
other match {
  // ...
}
```

33. У выражения сопоставления должно быть два варианта. В первом варианте должен объявляться типизированный шаблон для типа класса, в котором переопределяется метод `equals`:

```
case that: Rational =>
```

34. В теле этого варианта следует записать выражение, объединяющее с помощью логического И отдельные выражения, которые должны вычисляться в `true` для объектов, которые должны быть эквивалентными. Если переопределяемый метод `equals` отличается от метода в `AnyRef`, то, скорее всего, возникнет желание включить вызов метода `equals` из родительского класса:

```
super.equals(that) &&
```

Если определяется метод `equals` для класса, в котором впервые появляется метод `canEqual`, нужно вызвать `canEqual` в

отношении аргумента метода эквивалентности, передав в качестве аргумента ключевое слово `this`:

```
(that canEqual this) &&
```

В переопределения `equals` следует также включать вызов `canEqual`, если только в них не содержится вызов `super.equals`. В последнем варианте проверка с помощью `canEqual` уже должна выполняться путем вызова родительского класса. И наконец, для каждого поля, имеющего отношение к эквивалентности, следует проверить, что поле в объекте `this` эквивалентно соответствующему полю в переданном объекте:

```
numer == that.numer &&
denom == that.denom
```

35. Для второго варианта следует воспользоваться универсальным шаблоном сопоставления, выдающим `false`:

```
case _ => false
```

Если придерживаться этого рецепта для `equals`, равенство гарантированно будет отношением эквивалентности в соответствии с требованием соглашения по `equals`.

Рецепт для hashCode

Обычно удовлетворительные результаты для `hashCode` достигаются при использовании следующего рецепта, похожего на рецепты, рекомендованные для классов Java в книге Effective Java[145]. Возьмите в расчет все поля в своем объекте, которые используются для определения эквивалентности в методе `equals` (значимые поля). Создайте кортеж, содержащий значения всех этих полей. Затем вызовите для получившегося кортежа метод ##.

Например, для реализации хеш-кода для объекта, имеющего пять значимых полей с именами a, b, c, d и e, нужно сделать следующую запись:

```
override def hashCode: Int = (a, b, c, d, e).##
```

В том случае если метод equals в качестве части своих вычислений вызывает super.equals(that), вычисление в вашем методе hashCode должно начинаться с вызова super.hashCode.

Например, если в методе equals класса Rational вызывается super.equals(that), его метод hashCode должен иметь следующий вид:

```
override def hashCode: Int = (super.hashCode,
numer, denom).##
```

При написании методов hashCode с использованием данного подхода следует иметь в виду, что ваш хеш-код будет не хуже хеш-кодов, создаваемых за его пределами, а именно тех хеш-кодов, которые получаются путем вызова hashCode в отношении значимых полей вашего объекта. Иногда для получения приемлемого хеш-кода для такого поля одного только вызова hashCode в отношении этого поля может оказаться недостаточно. Например, если одно из полей является коллекцией, то, скорее всего, для него понадобится хеш-код, основанный на всех содержащихся в коллекции элементах. Когда поле относится к типу Vector, List, Set, Map или является кортежем, его можно просто включить в список хешируемых элементов, поскольку equals и hashCode переопределены в этих классах таким образом, чтобы брать в расчет содержащиеся в них элементы. Этого нельзя сказать о полях, относящихся к типу Array, в которых при вычислении хеш-кода элементы в расчет не берутся. Что касается массивов, то тут нужно рассматривать каждый элемент массива как отдельное поле вашего объекта, вызывая оператор ## в отношении каждого элемента в явном виде или передавая массив одному из методов

hashCode в синглтон-объекте java.util.Arrays.

И наконец, если обнаружится, что конкретно взятое вычисление хеш-кода отрицательно влияет на производительность вашей программы, рассмотрите вариант кэширования хеш-кода. Если объект неизменяемый, можно вычислить хеш-код при создании объекта и сохранить его в поле. Сделать это можно простым переопределением hashCode с указанием val вместо def:

```
override val hashCode: Int = (numer, denom).##
```

Этот подход связан с дополнительным расходом памяти в процессе вычисления, поскольку каждый экземпляр неизменяемого класса будет иметь на одно поле больше для сохранения кэшируемого значения хеш-кода.

Резюме

Оглядываясь назад, можно сказать, что правильная реализация equals оказалась на удивление непростым занятием. Нужно было проявлять осмотрительность в отношении сигнатуры типа, переопределять hashCode, обходить зависимости от изменяемого состояния и, если класс был нетерминальным, заниматься реализацией и использованием метода canEqual.

Столкнувшись со сложностями реализации корректного метода равенства, можно отдать предпочтение определению классов сопоставимых объектов в виде case-классов. Тогда компилятор Scala автоматически добавит методы equals и hashCode, имеющие нужные свойства.

138 Vaziri M., Tip F., Fink S., Dolby J. Declarative Object Identity Using Relation Types. — In Proc. ECOOP 2007, 2007. — P. 54–78.

139 Все просчеты, за исключением третьего, рассмотрены в контексте Java в книге Bloch J. Effective Java Second Edition. — Addison-Wesley, 2008.

140 При наличии большой практики программирования на Java вы, возможно, ожидали, что аргумент для этого метода будет относиться к типу Object, а не к Any. Напрасно

беспокоитесь: это все тот же метод equals. Компилятор просто делает вид, что он относится к типу Any.

141 Причиной разработки текста соглашения по определению в классе Any метода hashCode стала документация Javadoc, касающаяся класса java.lang.Object.

142 Как и в случае с hashCode, имеющееся в Any соглашение относительно метода equals основано на соглашении по методу equals, определяемому в java.lang.Object.

143 Благодаря новой версии equals в Point эта версия ColoredPoint больше не нарушает требования симметричности.

144 *Bloch J*. Effective Java Second Edition. — Addison-Wesley, 2008.

145 Bloch J. Effective Java Second Edition. — Addison-Wesley, 2008.

31. Сочетание кодов Scala и Java

Зачастую код Scala используется в тандеме с довольно объемными программами и средами на Java. Поскольку Scala обладает высокой степенью совместимости с Java, в большинстве случаев удается вполне успешно без особой оглядки сочетать два языка. Например, доподлинно известно, что со Scala вполне успешно работают такие стандартные среды, как Swing, Servlets и JUnit. И тем не менее иногда при сочетании Java и Scala можно столкнуться с некоторыми проблемами.

В этой главе описываются два аспекта объединения Java и Scala. Сначала мы рассмотрим вопрос перевода Scala в Java, приобретающий особую важность, если код Scala вызывается из Java. Затем разберем, как применять аннотации Java в Scala, играющие важную роль при необходимости использования Scala с имеющимися средами Java.

31.1. Использование Scala из Java

В основном о Scala можно размышлять на уровне исходного кода. Но более полное представление о работе системы можно получить, разобравшись с трансляцией этого кода. Более того, если код Scala вызывается из Java, следует знать, как именно код Scala выглядит с точки зрения Java.

Основные правила

Scala реализуется в виде трансляции в стандартные байт-коды Java. Насколько это возможно, средства Scala отображаются непосредственно на эквивалентные им средства Java. Например, классы, методы, строки и исключения Scala компилируются в тот же самый байт-код Java, что и их Java-аналоги.

Для этого иногда требовалось принимать трудные решения

относительно конструкции Scala. Например, было бы неплохо реализовывать разрешение перегружаемых методов на этапе выполнения программы, а не на этапе ее компиляции. Но такая конструкция нарушала бы положения, принятые в Java, существенно усложняя смешивание кода Java и Scala. В данном случае Scala придерживается разрешения перегрузки, принятого в Java, благодаря чему методы Scala и вызовы методов могут напрямую отображаться на методы и вызовы методов Java.

Для других свойств в Scala имеется своя собственная конструкция. Например, эквивалента трейтам в Java не существует. Аналогично, хотя обобщенные типы есть как в Scala, так и в Java, их особенности в этих системах сильно разнятся. Для подобных свойств языка код Scala не может быть напрямую отображен на конструкцию Java, следовательно, он должен быть преобразован с использованием каких-либо комбинаций имеющихся в Java структур.

Для таких косвенно отображаемых свойств преобразование не имеет фиксированных форм. Максимальное упрощение подобных преобразований требует постоянных усилий, поэтому на момент чтения данной книги некоторые особенности, существовавшие во времена ее написания, могли измениться. Какие именно преобразования использует ваш компилятор Scala, можно определить, изучив файлы с расширением `.class` с помощью инструментального средства, подобного `javap`.

Таковы общие правила. А теперь рассмотрим некоторые особые случаи.

Типы значений

Такой тип значения, как `Int`, может быть преобразован в Java двумя различными способами. При первой же возможности для получения наивысшей производительности компилятор преобразует Scala `Int` в Java `int`. Но иногда этого сделать не удается, поскольку компилятор не уверен, каким именно

преобразованием он занимается, `Int` или какого-либо другого типа данных. Например, конкретный `List[Any]` может содержать только `Int`-значения, но у компилятора нет возможности убедиться в этом.

Когда компилятор не уверен, относится объект к типу значений или нет, он использует объекты и полагается на классы-оболочки. Например, такие классы-оболочки, как `java.lang.Integer`, позволяют типу значений быть заключенным в Java-объект, в силу чего обрабатываться кодом, требующим объекты[146].

Синглтон-объекты

В Java нет точного эквивалента синглтон-объекту, но в нем имеются статические методы. В выполняемом Scala преобразовании синглтон-объектов используется сочетание статических методов и методов экземпляра. Для каждого синглтон-объекта Scala компилятор создаст класс Java с именем, соответствующим имени объекта, но со знаком доллара в конце. Например, для синглтон-объекта по имени `App` компилятор создаст класс Java по имени `App$`. Этот класс будет иметь все методы и поля синглтон-объекта Scala. В классе Java также будет одно статическое поле по имени `MODULE$`, предназначенное для хранения одного экземпляра класса, создаваемого на этапе выполнения программы.

Чтобы привести полноценный пример, предположим, что компилируется следующий синглтон-объект:

```
object App {
  def main(args: Array[String]) = {
    println("Hello, world!")
  }
}
```

Scala создаст Java-класс App$ со следующими полями и методами:

```
$ javap App$
public final class App$ extends java.lang.Object
implements scala.ScalaObject{
    public static final App$ MODULE$;
    public static {};
    public App$();
    public void main(java.lang.String[]);
    public int $tag();
}
```

Это общий вариант преобразования. Важным особым случаем является наличие обособленного синглтон-объекта, для которого не имеется класса с таким же именем. Например, у вас может быть синглтон-объект по имени App и не быть класса с именем App. В таком случае компилятор создаст Java-класс по имени App, имеющий статический метод перенаправления к каждому методу синглтон-объекта Scala:

```
$ javap App
Compiled from "App.scala"
public final class App extends java.lang.Object{
    public static final int $tag();
                public    static    final    void
main(java.lang.String[]);
}
```

В отличие от этого, если имелся класс по имени App, Scala создаст соответствующий Java-класс App для хранения элементов определенного вами класса App. В этот класс не будут добавляться методы перенаправления для синглтон-объектов с таким же именем, и код Java должен будет обращаться к синглтон-объекту

посредством поля `MODULE$`.

Трейты в качестве интерфейсов. При компиляции любого трейта создается Java-интерфейс с таким же именем. Этот интерфейс можно использовать как тип Java, и он позволяет вам вызывать методы в отношении объектов Scala через переменные этого типа.

Реализация трейта в Java — совсем другая история. Вообще-то это не практикуется, но один особый случай все же имеет важное значение. Если создается Scala-трейт, включающий только абстрактные методы, он будет напрямую преобразован в Java-интерфейс без какого-либо кода, за который стоило бы переживать. По сути, это означает, что при желании можно создавать интерфейс Java в синтаксисе Scala.

31.2. Аннотации

Основная система аннотаций Scala была рассмотрена в главе 27. В этом разделе описаны те свойства аннотаций, которые имеют непосредственное отношение к Java.

- **Дополнительные эффекты от стандартных аннотаций.** Некоторые аннотации заставляют компилятор выдавать дополнительную информацию с прицелом на платформу Java. Когда компилятору встречается такая аннотация, он сначала обрабатывает ее в соответствии с общими правилами, действующими в Scala, а затем выполняет дополнительные действия для Java.

- **Нежелательность.** Для любого метода или класса с меткой `@deprecated` компилятор добавит к выдаваемому коду аннотацию нежелательности, принадлежащую Java. Поэтому компиляторы Java могут выдавать предупреждения о нежелательности при обращении кода на Java к нежелательным методам Scala.

- **Изменяемые поля.** По аналогии с предыдущим описанием любое поле с меткой `@volatile` в коде Scala получает в выдаваемом коде Java-модификатор `volatile`. Благодаря этому изменяемые поля в Scala ведут себя в точном соответствии с семантикой Java и обращение к изменяемым полям выполняется в последовательности, точно соответствующей правилам, определенным для изменяемых полей в модели памяти Java.

- **Сериализация.** Все три имеющихся в Scala стандарта аннотирования сериализации преобразуются в их Java-эквиваленты. К классу с меткой `@serializable` добавляется Java-интерфейс `Serializable`. Аннотация `@SerialVersionUID(1234L)` превращается в следующее определение поля Java:

  ```
  // Используемый в Java маркер серийного номера версии
  private final static long SerialVersionUID = 1234L
  ```

  Любая переменная с меткой `@transient` получает Java-модификатор `transient`.

- **Выданные исключения.** В Scala перехват выданных исключений не контролируется. Поэтому в Scala нет эквивалента имеющемуся в Java объявлению методов с ключевым словом `throws`. Все методы Scala переводятся в методы Java, в которых не объявляется выдача исключений[147].

Последнее свойство было убрано из Scala потому, что опыт работы с ним, накопленный в Java, имеет несколько негативный оттенок. Поскольку аннотирование методов с помощью `throws` вызывает трудности, многие разработчики создают код,

подавляющий и сбрасывающий исключения, просто с целью получения кода для компиляции без добавления всех этих `throws`-описаний. У них может быть намерение улучшить систему обработки исключений чуть позже, но опыт показывает, что ни один из зачастую ограниченных во времени программистов никогда не вернется назад и не добавит соответствующую обработку исключений. Обратная сторона медали выглядит так, что это реализуемое с благими намерениями свойство зачастую снижает надежность кода. Огромная масса выработанного кода Java подавляет и скрывает выдачу исключений на этапе выполнения программ, дабы удовлетворить компилятор.

Но иногда при разработке интерфейса к Java может потребоваться создать код Scala, имеющий дружественные по отношению к Java аннотации с описаниями того, какие именно исключения могут выдавать ваши методы. К примеру, каждый метод в удаленном интерфейсе RMI требует в его описании `throws` упоминания о `java.io.RemoteException`. То есть, если нужно создать удаленный интерфейс RMI в виде Scala-трейта с абстрактными методами, то для этих методов нужно будет в описании `throws` перечислить `RemoteException`. Для этого понадобится всего лишь пометить свои методы аннотацией `@throws`. Например, в Scala-классе, показанном в листинге 31.1, имеется метод с меткой, сообщающей о том, что он выдает исключение `IOException`.

Листинг 31.1. Метод Scala, объявляющий Java-описание throws

```
import java.io._
class Reader(fname: String) {
  private val in =
    new BufferedReader(new FileReader(fname))
  @throws(classOf[IOException])
  def read() = in.read()
```

```
}
```

А вот как это будет выглядеть со стороны Java:

```
$ javap Reader
Compiled from "Reader.scala"
public class Reader extends java.lang.Object
implements
scala.ScalaObject{
    public Reader(java.lang.String);
    public int read() throws java.io.IOException;
    public int $tag();
}
$
```

Заметьте, что метод `read` показывает наряду с Java-объявлением `throws`, что может быть выдано исключение `IOException`.

Аннотации Java

Аннотации, имеющиеся в среде Java, могут использоваться в коде Scala напрямую. Они будут видны любой Java-среде точно так же, как будто были написаны в коде Java.

Аннотации используются множеством различных пакетов Java. Рассмотрим в качестве примера JUnit 4. JUnit является средой для написания и запуска автоматизированных тестов. Самая последняя версия, JUnit 4, использует аннотации, чтобы показать, какие части кода тестируются. Замысел состоит в том, что вы написали для своего кода множество тестов, которые запускаются после каждого изменения исходного кода. Поэтому, если при изменении допущена новая ошибка, один из тестов не будет пройден и ошибка тут же обнаружится.

Написать тест не составляет труда. Просто в классе верхнего

уровня создается метод, выполняющий ваш код, а для того, чтобы обозначить его как тест, используется аннотация. Выглядит это следующим образом:

```
import org.junit.Test
import org.junit.Assert.assertEquals

class SetTest {

  @Test
  def testMultiAdd = {
    val set = Set() + 1 + 2 + 3 + 1 + 2 + 3
    assertEquals(3, set.size)
  }
}
```

Метод `testMultiAdd` является тестом. Он добавляет несколько элементов к набору и убеждается в том, что каждый из них был добавлен только один раз. Метод `assertEquals`, получаемый в составе JUnit API, проверяет, что два его аргумента равны друг другу. Если они отличаются друг от друга, тест пройден не будет. В таком случае тест проверяет, что повторно добавляемые одинаковые числа не увеличили размер набора.

Тест помечается с использованием аннотации `org.junit.Test`. Заметьте, что эта аннотация была импортирована, следовательно, на нее можно сослаться просто как на `@Test`, не прибегая к более длинной ссылке `@org.junit.Test`.

Вот, собственно, и все. Тест может быть запущен с использованием любого средства выполнения тестов JUnit. Здесь он запускается средством, работающим в командной строке:

```
$ scala -cp junit-4.3.1.jar:. org.junit.runner.JUnitCore SetTest
JUnit version 4.3.1
```

```
.
Time: 0.023

OK (1 test)
```

Написание собственных аннотаций

Чтобы создать аннотацию, которая будет видна Java-рефлексии, нужно воспользоваться принятой в Java системой записи и откомпилировать аннотацию с помощью `javac`. В данном случае вряд ли есть смысл записывать аннотации в Scala, поскольку стандартный компилятор их не поддерживает. Дело в том, что поддержка Scala никак не будет соответствовать полным возможностям аннотаций Java, а кроме того, когда-нибудь в Scala появится собственная рефлексия, и тогда захочется получить доступ к аннотациям Scala со Scala-рефлексией.

Пример аннотации:

```
import java.lang.annotation. * ; // Это код Java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Ignore { }
```

После компиляции показанного ранее кода с применением `javac` аннотацией можно будет воспользоваться следующим образом:

```
object Tests {
  @Ignore
  def testData = List(0, 1, -1, 5, -5)

  def test1 = {
          assert(testData  ==  (testData.head  ::
testData.tail))
```

```
  }

  def test2 = {
    assert(testData.contains(testData.head))
  }
}
```

В этом примере предполагается, что `test1` и `test2` являются тестовыми методами, а `testData` должен игнорироваться, несмотря на то что его имя начинается с `test`.

Чтобы убедиться в наличии данной аннотации, можно воспользоваться API рефлексии Java. Для отражения этого в работе можно воспользоваться следующим примером:

```
for {
  method <- Tests.getClass.getMethods
  if method.getName.startsWith("test")
  if method.getAnnotation(classOf[Ignore]) == null
} {
  println("found a test method: " + method)
}
```

Здесь рефлексивные методы `getClass` и `getMethods` используются для инспектирования всех полей класса вводимого объекта. Это обычные рефлексивные методы. В той части, которая относится непосредственно к аннотации, используется метод `getAnnotation`. Этот метод, имеющийся у многих рефлексивных объектов, предназначен для поиска аннотаций конкретного типа. В данном случае код ищет аннотацию нашего нового типа `Ignore`. Поскольку это API, успешное выполнение выявляется по возвращаемому результату, который может быть либо `null`, либо текущим объектом аннотации.

Рассмотрим работу этого кода:

```
$ javac Ignore.java
$ scalac Tests.scala
$ scalac FindTests.scala
$ scala FindTests
found a test method: public void Tests$.test2()
found a test method: public void Tests$.test1()
```

Попутно обратите внимание на то, что эти методы при просмотре в Java-рефлексии относятся к классу `Tests$`, а не `Tests`. В начале главы уже говорилось, что в Java реализация синглтон-объекта Scala помещается в Java-класс с добавлением в конец имени этого класса знака доллара. В данном случае реализация `Tests` попадает в Java-класс `Tests$`.

Следует понимать, что при использовании аннотаций Java придется работать в рамках существующих для них ограничений. Например, в аргументах к аннотациям можно использовать только константы, а не выражения. Может поддерживаться `@serial(1234)`, но никак не `@serial(x * 2)`, потому что `x * 2` не является константой.

31.3. Подстановочные типы

У всех типов Java имеется Scala-эквивалент. Это нужно, чтобы код Scala мог обращаться к любому допустимому в Java классу. В большинстве случаев преобразование не вызывает никаких затруднений. `Pattern` в Java — это `Pattern` в Scala, а `Iterator<Component>` в Java — это `Iterator[Component]` в Scala. Но в некоторых случаях встречавшихся до сих пор типов Scala оказывается недостаточно. Что, к примеру, можно сделать с такими подстановочными типами Java (wildcard types), как `Iterator<?>` или `Iterator<? extends Component>`? А что можно сделать с такими необработанными типами, как `Iterator`, где параметр типа опущен? Для подстановочных типов Java и необработанных типов в Scala используется дополнительная

разновидность типа, которая также называется *подстановочным типом*.

Подстановочные типы записываются с применением синтаксиса заместителя, что очень похоже на краткую форму записи функционального литерала, рассмотренную в разделе 8.5. В сокращении для функциональных литералов вместо выражения можно использовать символ подчеркивания (_). Например, `(_ + 1)` является аналогом `(x => x + 1)`. В подстановочных типах применяется тот же принцип, но только для типов, а не для выражений. Если написать `Iterator[_]`, то знак подчеркивания замещает тип. Такой тип представляет собой `Iterator` с неизвестным типом элементов.

При использовании синтаксиса заместителей можно также вставлять нижние и верхние ограничители. Нужно просто добавить ограничитель после знака подчеркивания, используя тот же самый синтаксис `<:`, который применялся для параметров типа (см. разделы 19.5 и 19.8). Например, тип `Iterator[_ <: Component]` является итератором с неизвестным типом элементов, но каким бы ни был этот тип, он должен быть подтипом типа `Component`.

С записью подстановочного типа разобрались, а как насчет его использования? В простых случаях можно игнорировать подстановку и вызывать методы в отношении базового типа. Предположим, к примеру, что у вас имеется следующий Java-класс:

```
// Это Java-класс с подстановками
public class Wild {
  public Collection<?> contents() {
     Collection<String> stuff = new Vector<String>
();
    stuff.add("a");
    stuff.add("b");
    stuff.add("see");
```

```
    return stuff;
  }
}
```

Если обращаться к этому коду из кода Scala, будет видно, что у него имеется подстановочный тип:

```
scala> val contents = (new Wild).contents
contents: java.util.Collection[_] = [a, b, see]
```

Если нужно определить количество элементов в коллекции, можно просто проигнорировать подстановочную часть и вызвать обычным способом метод `size`:

```
scala> contents.size()
res0: Int = 3
```

В более сложных случаях подстановочные типы могут создавать проблемы посерьезнее. Поскольку у подстановочного типа нет имени, то нет и способа использовать его в двух разных местах. Предположим, к примеру, что нужно создать изменяемый Scala-набор и инициализировать его элементами `contents`:

```
import scala.collection.mutable
val iter = (new Wild).contents.iterator
val set = mutable.Set.empty[???] // Какой тип сюда попадет?
while (iter.hasMore)
  set += iter.next()
```

Проблема возникает в третьей строке. В Java-коллекции назвать тип элементов невозможно, следовательно, нельзя удовлетворительным образом записать тип набора. Чтобы обойти данную проблему, имеются два приема, требующих рассмотрения.

При передаче подстановочного типа в метод задайте методу

параметр для заместителя. Теперь у вас есть название типа, которым можно воспользоваться желаемое количество раз.

36. Вместо возвращения из метода подстановочного типа возвращайте объект, имеющий абстрактные элементы для каждого типа, обозначенного заместителем. (Сведения об абстрактных элементах можно найти в главе 20.)

Используя вместе эти два приема, предыдущий код можно превратить в следующий:

```
import scala.collection.mutable
import java.util.Collection

abstract class SetAndType {
  type Elem
  val set: mutable.Set[Elem]
}

def javaSet2ScalaSet[T](jset: Collection[T]):
SetAndType = {
   val sset = mutable.Set.empty[T] // теперь T
может быть назван!

  val iter = jset.iterator
  while (iter.hasNext)
    sset += iter.next()

  return new SetAndType {
    type Elem = T
    val set = sset
  }
}
```

Можно понять, почему в коде Scala подстановочные типы обычно не используются. Чтобы не возиться с ними, многие склоняются к преобразованию этих типов в абстрактные элементы. Поэтому для начала можно также воспользоваться абстрактными элементами.

31.4. Совместная компиляция Scala и Java

Обычно при проведении компиляции кода Scala, который зависит от кода Java, сначала создается Java-код файлов класса. Затем создается код Scala, а файлы с классами в коде Java указываются в пути к классам. Но подобный подход не будет работать, если в коде Java есть обратные ссылки на код Scala. В таком случае порядок, в котором компилируется код, не играет никакой роли — либо та, либо другая сторона будет иметь неудовлетворенные внешние ссылки. Такая ситуация не является редкостью, она возникает главным образом в Java-проекте, где какой-нибудь файл исходного кода на Java заменяется файлом исходного кода на Scala.

Для поддержки подобных сборок Scala позволяет компилировать исходный код Java, так же как и файлы классов Java. Нужно лишь указать файлы с исходным кодом на Java в командной строке, как будто это файлы Scala. Компилятор Scala не станет компилировать эти Java-файлы, но он их просканирует, чтобы понять, что в них содержится. Чтобы воспользоваться этим средством, сначала компилируется Scala-код с использованием файлов исходного кода на Java, а затем Java-код с использованием файлов классов Scala.

А вот как выглядит типичная последовательность команд:

```
$ scalac -d bin InventoryAnalysis.scala
InventoryItem.java \
      Inventory.java
$ javac -cp bin -d bin Inventory.java
InventoryItem.java \
```

```
        InventoryManagement.java
$ scala -cp bin InventoryManagement
Most expensive item = sprocket($4.99)
```

31.5. Интегрирование Java 8 в Scala 2.12

В версии Java 8 в язык Java и в байт-коды был добавлен ряд усовершенствований, которыми пользуется Scala в своем выпуске 2.12148. Используя новые возможности Java 8, компилятор Scala 2.12 может создавать более компактные файлы классов и jar-файлы и улучшить двоичную совместимость трейтов.

Лямбда-выражения и SAM-типы

С точки зрения программистов, работающих на Scala, наиболее ощутимые усовершенствования в Scala 2.12, связанные с применением Java 8, заключаются в том, что функциональные литералы Scala могут использоваться подобно *лямбда-выражениям* в качестве более краткой формы выражения экземпляров безымянных классов. Чтобы передать поведение в метод до выхода Java 8, Java-программисты зачастую определяли безымянные внутренние экземпляры класса:

```
JButton button = new JButton(); // Это код на Java
button.addActionListener(
  new ActionListener() {
    public void actionPerformed(ActionEvent event)
{
      System.out.println("pressed!");
    }
  }
);
```

В этом примере создается безымянный экземпляр

ActionListener, который передается принадлежащему JButton среды Swing методу addActionListener. Когда пользователь щелкает на кнопке, среда Swing вызывает в отношении этого экземпляра метод actionPerformed, который выводит сообщение "pressed!".

В Java 8 лямбда-выражение может использоваться везде, где требуется экземпляр класса или интерфейса, содержащий единственный абстрактный метод (single abstract method (SAM)). ActionListener является именно таким интерфейсом, поскольку содержит единственный абстрактный метод actionPerformed. Следовательно, лямбда-выражение может использоваться для регистрации отслеживателя действия в отношении Swing-кнопки, например:

```
JButton button = new JButton(); // это код Java 8
button.addActionListener(
  event -> System.out.println("pressed!")
);
```

В Scala в подобной ситуации также можно использовать безымянный экземпляр внутреннего класса, но можно и отдать предпочтение применению функционального литерала:

```
val button = new JButton
button.addActionListener(
  _ => println("pressed!")
)
```

Как было показано в разделе 21.1, подобный стиль программирования можно поддерживать за счет определения подразумеваемого преобразования из функционального типа ActionEvent => Unit в ActionListener.

В Scala 2.12 в таком случае разрешается использовать функциональный литерал даже при отсутствии такого

подразумеваемого преобразования. Как и в Java 8, в Scala 2.12 позволяется использовать функциональный тип там, где требуется экземпляр класса или трейта, декларирующий единственный абстрактный метод (SAM). В Scala 2.12 это работает с любым SAM-типом. Например, можно определить трейт `Increaser` с единственным абстрактным методом `increase`:

```
scala> trait Increaser {
         def increase(i: Int): Int
       }
defined trait Increaser
```

Затем можно определить метод, получающий `Increaser`:

```
scala> def increaseOne(increaser: Increaser): Int
=
         increaser.increase(1)
increaseOne: (increaser: Increaser)Int
```

Для вызова вашего нового метода можно передать безымянный экземпляр трейта `Increaser`:

```
scala> increaseOne(
         new Increaser {
           def increase(i: Int): Int = i + 7
         }
       )
res0: Int = 8
```

Но в Scala 2.12 в качестве альтернативы можно просто воспользоваться функциональным литералом, поскольку `Increaser` относится к SAM-типу:

```
scala> increaseOne(i => i + 7) // Scala 2.12
res1: Int = 8
```

Использование Stream-объектов Java 8 из Scala 2.12

Stream в Java является функциональной структурой данных, предоставляющей метод map, принимающий java.util.function.IntUnaryOperator. Из Scala Stream.map можно вызвать для увеличения значения каждого элемента Array-объекта на единицу:

```
scala> import java.util.function.IntUnaryOperator
import java.util.function.IntUnaryOperator

scala> import java.util.Arrays
import java.util.Arrays

scala> val stream = Arrays.stream(Array(1, 2, 3))
stream: java.util.stream.IntStream = ...

scala> stream.map(
         new IntUnaryOperator {
           def applyAsInt(i: Int): Int = i + 1
         }
       ).toArray
res3: Array[Int] = Array(2, 3, 4)
```

Но поскольку IntUnaryOperator относится к SAM-типу, в Scala 2.12 его можно вызвать в более краткой форме с помощью функционального литерала:

```
scala> val stream = Arrays.stream(Array(1, 2, 3))
stream: java.util.stream.IntStream = ...

scala> stream.map(i => i + 1).toArray // Scala 2.12
res4: Array[Int] = Array(2, 3, 4)
```

Заметьте, что к SAM-типам могут быть адаптированы только функциональные литералы, а не произвольные выражения, имеющие функциональный тип. Рассмотрим, к примеру, следующую `val`-переменную `f`, имеющую тип `Int => Int`:

```
scala> val f = (i: Int) => i + 1
f: Int => Int = ...
```

Хотя у `f` тот же тип, что и у функционального литерала, переданного ранее `stream.map`, использовать `f` там, где требуется `IntUnaryOperator`, нельзя:

```
scala> val stream = Arrays.stream(Array(1, 2, 3))
stream: java.util.stream.IntStream = ...

scala> stream.map(f).toArray
<console>:16: error: type mismatch;
found : Int => Int
required: java.util.function.IntUnaryOperator
        stream.map(f).toArray
                   ^
```

Чтобы воспользоваться `f`, можно явным образом указать вызов, используя функциональный литерал:

```
scala> stream.map(i => f(i)).toArray
res5: Array[Int] = Array(2, 3, 4)
```

Или же можно проаннотировать `f` при определении с помощью `IntUnaryOperator`, то есть того типа, который ожидается `Stream.map`:

```
scala> val f: IntUnaryOperator = i => i + 1
f: java.util.function.IntUnaryOperator = ...
```

```
scala> val stream = Arrays.stream(Array(1, 2, 3))
stream: java.util.stream.IntStream = ...

scala> stream.map(f).toArray
res6: Array[Int] = Array(2, 3, 4)
```

При работе со Scala 2.12 и Java 8 можно также, используя лямбда-выражения Java, вызывать из Java методы, скомпилированные Scala, передавая функциональные типы Scala. Хотя функциональные типы Scala определяются как трейты, включающие конкретные методы, Scala 2.12 компилирует трейты в Java-интерфейсы с методами по умолчанию, которые являются новой особенностью Java 8. В результате функциональные типы Scala оказываются в Java SAM-типами.

Резюме

В большинстве случаев способ реализации Scala можно игнорировать и просто создавать и запускать на выполнение свой код. Но иногда бывает полезно, что называется, заглянуть под капот, поэтому в данной главе рассматривались три аспекта реализации Scala на платформе Java: на что похоже преобразование кода, как аннотации Scala и Java работают вместе и как подстановочные типы Scala позволяют обращаться к подстановочным типам Java. Кроме этого, рассматривались использование из Scala имеющихся в Java примитивов многопоточных вычислений и компиляция объединенных Scala- и Java-проектов. Особо важное значение эти темы приобретают при совместном использовании Scala и Java.

146 Реализация типов значений была подробно рассмотрена в разделе 11.2.

147 Код все равно работает, поскольку верификатор байт-кода Java вообще не проверяет объявления! Этим занимается компилятор Java.

148 Чтобы в Scala можно было воспользоваться свойствами Java 8, для Scala 2.12 требуется Java 8.

32. Фьючерсы и многопоточные вычисления

Одним из результатов распространения многоядерных процессоров стал повышенный интерес к многопоточным вычислениям. В Java поддержка многопоточных вычислений выстроена вокруг совместно используемой памяти и блокировок. При всей своей достаточности этот подход получился слишком сложным для практической реализации. В стандартной библиотеке Scala предлагается альтернатива, обходящая эти трудности за счет того, что основное внимание уделяется `Future`-технологии, представляющей собой асинхронные преобразования неизменяемого состояния.

Хотя в Java также предлагается фьючерс — `Future`, его вариант сильно отличается от того, что используется в Scala. Оба они представляют результат асинхронного вычисления, но `Future` в Java требует обращения к результату через блокирующий метод `get`. Несмотря на то что в Java, чтобы узнать о завершении `Future` перед вызовом `get`, можно вызвать метод `isDone`, избавляясь тем самым от применения любых блокировок, перед выполнением каких бы то ни было вычислений, использующих результат, нужно дождаться завершения Java `Future`.

В отличие от этого, преобразования Scala `Future` можно указать независимо от того, завершено соответствующее вычисление или нет. Каждое преобразование приводит к созданию нового `Future`-экземпляра, представляющего асинхронный результат исходного `Future`-экземпляра, преобразованный функцией. Поток, выполняющий вычисление, определяется подразумеваемо предоставляемым *контекстом выполнения*. Это позволяет давать описание асинхронных вычислений в виде серий преобразований неизменяемых значений, не ощущая при этом потребности в совместном использовании памяти и в блокировках.

32.1. Неприятности в раю

Каждый объект на платформе Java связан с логическим монитором, который можно использовать для управления многопоточным доступом к данным. Для применения этой модели следует решить, какие именно данные будут совместно использоваться несколькими потоками, и пометить как синхронизируемые (`synchronized`) те разделы кода, которые обращаются к совместно используемым данным или управляют доступом к ним. Система выполнения Java применяет механизм блокировок, чтобы гарантировать, что в определенный момент времени в синхронизируемые разделы, охраняемые отдельно взятой блокировкой, войдет только один поток, позволяя тем самым дирижировать многопоточным доступом к совместно используемым данным.

Из соображений совместимости в Scala предоставляется доступ к имеющимся в Java примитивам многопоточных вычислений. В Scala могут вызываться методы `wait`, `notify` и `notifyAll`, и у них точно такое же предназначение, как и в Java. Технически ключевого слова `synchronized` в Scala нет, но в этот язык включены предопределенные методы синхронизации, вызываемые следующим образом:

```
var counter = 0
synchronized {
  // этот код в определенный момент времени может
выполняться
  // только одним потоком
  counter = counter + 1
}
```

К сожалению, программистам было очень сложно создавать действительно надежные многопоточные приложения с помощью модели совместно используемых данных и блокировок, особенно с повышением объема и сложности приложений. Дело в том, что в

каждой точке программы требовалось размышлять о том, какие данные изменяются или к каким данным происходит обращение и могут ли к ним обращаться или вносить в их изменения другие потоки, удерживаемые от этих действий блокировкой. При каждом вызове метода требовалось понимать, какие именно средства блокировки он испробует для сдерживания, и нужно было убеждать самого себя, что при попытке их получения не возникнет взаимной блокировки. Задачу усложняет еще и то, что блокировки, о которых мы говорим, не фиксируются в ходе компиляции, поскольку программа на этапе выполнения может беспрепятственно создавать новые блокировки.

Усугубляет сложившуюся ситуацию то, что тестирование многопоточного кода не дает надежных результатов. Поскольку потоки имеют недетерминированный характер, программа может успешно пройти тестирование тысячу раз и сработать неподобающим образом при первом же запуске на машине клиента. Из-за этого при использовании общих данных и блокировок остается только уповать на корректность программы.

Более того, введением избыточной синхронизации проблема не решается. Уровень проблематичности повсеместного применения синхронизации может быть сопоставим с уровнем проблематичности полного отказа от синхронизации. Хотя новые блокировки могут устранить вероятность появления ситуаций соперничества, одновременно с этим они повышают риск возникновения взаимных блокировок. Корректная программа, рассчитанная на длительный период использования, должна исключать как ситуации соперничества потоков, так и ситуации взаимных блокировок, поэтому в манипуляциях с программой нельзя чересчур усердствовать в обоих направлениях.

Высокоуровневые абстракции для многопоточного программирования предоставляются библиотекой `java.util.concurrent`. Использование утилит многопоточных вычислений делает процесс многопоточного программирования гораздо менее подверженным возникновению ошибок по

сравнению с обкаткой ваших собственных абстракций, создаваемых на основе имеющихся в Java низкоуровневых примитивов синхронизации. И тем не менее утилиты многопоточных вычислений также основаны на применении модели совместно используемых данных и блокировок, что не позволяет преодолевать основные трудности, возникающие при использовании данной модели.

32.2. Асинхронное выполнение и Try

Хотя универсального решения не существует, один из путей, позволяющих справиться с многопоточным вычислением, предлагает существующая в Scala технология `Future`, которая может уменьшить, а зачастую и устранить необходимость рассматривать вопросы применения совместно используемых данных и блокировок. При вызове метода Scala он выполняет вычисление, пока вы ждете, и возвращает результат. Если этот результат имеет тип `Future`, экземпляр `Future` заявляет еще одно вычисление, выполняемое в асинхронном режиме, которое зачастую выполняет другой поток. В результате этого многие операции, проводимые над `Future`-объектом, требуют подразумеваемого *контекста выполнения*, который предоставляет стратегию для асинхронного выполнения функций. Например, при попытке создания фьючерса посредством фабричного метода `Future.apply` без предоставления подразумеваемого контекста выполнения, то есть экземпляра `scala.concurrent.ExecutionContext`, будет получена ошибка компиляции:

```
scala> import scala.concurrent.Future
import scala.concurrent.Future

scala> val fut = Future { Thread.sleep(10000); 21
+ 21 }
```

```
<console>:11: error: Cannot find an implicit
ExecutionContext.
           You might pass an (implicit ec:
ExecutionContext)
    parameter to your method or import
    scala.concurrent.ExecutionContext.Implicits.glo
        val fut = Future { Thread.sleep(10000); 21
+ 21 }
                           ^
```

Сообщение об ошибке подсказывает один из путей решения проблемы: импорт глобального контекста выполнения, предоставляемого самим языком Scala. Глобальный контекст выполнения на виртуальной машине Java, JVM, использует пул потоков (thread pool)[149]. Как только подразумеваемый контекст выполнения будет введен в область видимости, можно будет создавать фьючерс:

```
scala> import
scala.concurrent.ExecutionContext.Implicits.global
import
scala.concurrent.ExecutionContext.Implicits.global

scala> val fut = Future { Thread.sleep(10000); 21
+ 21 }
fut: scala.concurrent.Future[Int] = ...
```

Фьючерс, созданный в предыдущем примере, выполняет в асинхронном режиме блок кода, используя глобальный контекст выполнения, затем завершается, выдавая значение 42. Как только он приступит к выполнению, данный поток станет спящим на 10 секунд. Таким образом, у данного фьючерса на завершение будет по крайней мере 10 секунд.

Выполнять опрос позволяют два метода класса `Future` — `isCompleted` и `value`. Вызов в отношении еще не завершившегося

фьючерса метода isCompleted приведет к возвращению значения false, а методом value при таких обстоятельствах будет возвращено значение None:

```
scala> fut.isCompleted
res0: Boolean = false

scala> fut.value
res1: Option[scala.util.Try[Int]] = None
```

Как только фьючерс будет завершен (в данном случае после того, как пройдет по крайней мере 10 секунд), метод isCompleted возвратит значение true, а метод value — значение типа Some:

```
scala> fut.isCompleted
res2: Boolean = true

scala> fut.value
res3: Option[scala.util.Try[Int]] = Some(Success(42))
```

В Option-значении, возвращенном value, содержится указание на тип Try. Как показано на рис. 32.1, Try выражается либо подклассом Success, содержащим значение T, либо подклассом Failure, содержащим исключение (экземпляр класса java.lang.Throwable). Предназначением Try является предоставление для асинхронных вычислений той же самой возможности, которую try-выражение предоставляло для синхронных вычислений: этот класс позволяет вам справляться с вероятностью внезапного завершения вычисления с выдачей исключения вместо возвращения результата[150].

Чтобы обеспечить для синхронных вычислений перехват и обработку выданных методом исключений тем самым потоком, который этот метод вызвал, можно воспользоваться инструкциями

try-catch. Но при асинхронных вычислениях поток, инициировавший вычисление, зачастую переключается на выполнение других задач. Чуть позже, если асинхронное вычисление даст сбой с выдачей исключения, исходный поток уже не сможет обработать исключение в блоке catch. Следовательно, чтобы справиться с возможным внезапным сбоем и выдачей исключения взамен итогового значения при работе с Future-экземпляром, выполняющим асинхронную работу, используется Try. Рассмотрим пример, показывающий, что получится при сбое асинхронной работы:

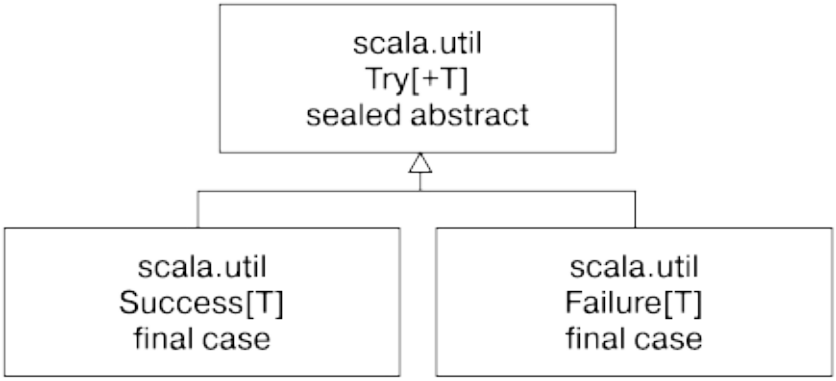


Рис. 32.1. Иерархия классов для Try

```
scala> val fut = Future { Thread.sleep(10000); 21
/ 0 }
fut: scala.concurrent.Future[Int] = ...

scala> fut.value
res4: Option[scala.util.Try[Int]] = None
```

Затем по прошествии 10 секунд:

```
scala> fut.value
res5: Option[scala.util.Try[Int]] =
     Some(Failure(java.lang.ArithmeticException: /
by zero))
```

32.3. Работа с фьючерсами

Фьючерсы в Scala позволяют определять преобразования Future-результатов и получать *новые фьючерсы*, представляющие собой композицию из двух асинхронных вычислений: исходного вычисления и преобразования.

Преобразование фьючерсов с помощью функции map

Основной операцией такого преобразования является map. Вместо блокировки при продолжении работы с другим вычислением можно просто отобразить следующее вычисление на фьючерс. Результатом станет новый фьючерс, представляющий исходный асинхронно вычисленный результат, асинхронно преобразованный функцией, переданной методу map.

Например, следующий фьючерс завершится по прошествии 10 секунд:

```
scala> val fut = Future { Thread.sleep(10000); 21
+ 21 }
fut: scala.concurrent.Future[Int] = ...
```

Отображение этого фьючерса с помощью функции, увеличивающей значение на единицу, приведет к выдаче еще одного фьючерса. Этот новый фьючерс будет представлять вычисление, состоящее из исходного сложения, за которым будет идти инкремент:

```
scala> val result = fut.map(x => x + 1)
result: scala.concurrent.Future[Int] = ...

scala> result.value
res5: Option[scala.util.Try[Int]] = None
```

Как только исходный фьючерс завершится и функция будет

применена к его результату, завершится и фьючерс, возвращенный методом map:

```
scala> result.value
res6:         Option[scala.util.Try[Int]]         =
Some(Success(43))
```

Следует заметить, что операции, намеченные для выполнения в данном примере, то есть создание фьючерса, вычисление суммы 21 + 21 и вычисление инкремента 42 + 1, могут быть выполнены тремя разными потоками.

Преобразование фьючерсов с помощью выражений for

Поскольку во фьючерсе Scala также объявляется метод flatMap, преобразование фьючерсов можно выполнять с помощью выражения for. Рассмотрим, к примеру, следующие два фьючерса, которые по прошествии 10 секунд выдадут 42 и 46:

```
scala> val fut1 = Future { Thread.sleep(10000); 21
+ 21 }
fut1: scala.concurrent.Future[Int] = ...

scala> val fut2 = Future { Thread.sleep(10000); 23
+ 23 }
fut2: scala.concurrent.Future[Int] = ...
```

На основе этих двух фьючерсов можно получить новый фьючерс, представляющий асинхронное суммирование их результатов:

```
scala> for {
         x <- fut1
         y <- fut2
       } yield x + y
```

```
res7: scala.concurrent.Future[Int] = ...
```

Как только завершатся исходный фьючерс и последующее суммирование, вы сможете увидеть результат:

```
scala> res7.value
res8: Option[scala.util.Try[Int]] =
Some(Success(88))
```

Поскольку выражение `for` выстраивает свои преобразования в последовательность[151], если не создать фьючерс до применения выражения `for`, преобразования не будут запущены в параллельных вычислениях. Например, хотя предыдущее выражение `for` требует для завершения около 10 секунд, следующее выражение `for` требует как минимум 20 секунд:

```
scala> for {
         x <- Future { Thread.sleep(10000); 21 + 21
}
         y <- Future { Thread.sleep(10000); 23 + 23
}
       } yield x + y
res9: scala.concurrent.Future[Int] = ...

scala> res9.value
res27: Option[scala.util.Try[Int]] = None

scala> // понадобится как минимум 20 секунд на
завершение
scala> res9.value
res28: Option[scala.util.Try[Int]] =
Some(Success(88))
```

Создание Future: Future.failed, Future.successful, Future.fromTry и

Promise

Кроме метода `apply`, использованного в приведенных ранее примерах для создания фьючерсов, объект-спутник `Future` включает также три фабричных метода для создания уже завершенных фьючерсов: `successful`, `failed` и `fromTry`. Эти методы не требуют контекста выполнения `ExecutionContext`.

Фабричный метод `successful` создает фьючерс, который уже успешно завершился:

```
scala> Future.successful { 21 + 21 }
res2: scala.concurrent.Future[Int] = ...
```

Метод `failed` создает фьючерс, который уже дал сбой:

```
scala> Future.failed(new Exception("bummer!"))
res3: scala.concurrent.Future[Nothing] = ...
```

Метод `fromTry` создает уже завершенный фьючерс из `Try`:

```
scala> import scala.util.{Success,Failure}
import scala.util.{Success, Failure}

scala> Future.fromTry(Success { 21 + 21 })
res4: scala.concurrent.Future[Int] = ...

scala>                    Future.fromTry(Failure(new
Exception("bummer!")))
res5: scala.concurrent.Future[Nothing] = ...
```

Наиболее универсальным способом создания фьючерса является использование обещания `Promise`. Давая обещание, можно получить фьючерс, управляемый обещанием. Фьючерс будет завершен, когда вы завершите обещание. Рассмотрим пример:

```
scala> val pro = Promise[Int]
pro: scala.concurrent.Promise[Int] = ...

scala> val fut = pro.future
fut: scala.concurrent.Future[Int] = ...
scala> fut.value
res8: Option[scala.util.Try[Int]] = None
```

Завершить обещание можно с помощью методов `success`, `failure` и `complete`. Эти методы, применяемые к `Promise`-объектам, аналогичны ранее рассмотренным, которые используются для создания уже завершенных фьючерсов. Например, метод `success` успешно завершит фьючерс:

```
scala> pro.success(42)
res9: pro.type = ...

scala> fut.value
res10:        Option[scala.util.Try[Int]]        =
Some(Success(42))
```

Метод `failure` получает исключение, заставляющее фьючерс выдать сбой с этим исключением. Метод `complete` получает `Try`. Также есть метод `completeWith`, получающий фьючерс. Состояние завершения того фьючерса, который был передан методу `completeWith`, впоследствии будет зеркально отображено в фьючерсе, принадлежащем обещанию.

Фильтрация: filter и collect

Фьючерсы Scala предлагают два метода, `filter` и `collect`, позволяющие гарантировать, что свойство истинно (`true`) относительно значения фьючерса. Метод `filter` проверяет результат фьючерса, оставляя его неизменным, если он

действителен. Пример, гарантирующий, что Int-значение является положительным, выглядит следующим образом:

```
scala> val fut = Future { 42 }
fut: scala.concurrent.Future[Int] = ...

scala> val valid = fut.filter(res => res > 0)
valid: scala.concurrent.Future[Int] = ...

scala> valid.value
res0: Option[scala.util.Try[Int]] =
Some(Success(42))
```

Если значение фьючерса недействительно, фьючерс, возвращаемый filter, даст сбой с выдачей исключения NoSuchElementException:

```
scala> val invalid = fut.filter(res => res < 0)
invalid: scala.concurrent.Future[Int] = ...

scala> invalid.value
res1: Option[scala.util.Try[Int]] =
  Some(Failure(java.util.NoSuchElementException:
  Future.filter predicate is not satisfied))
```

Поскольку в Future предлагается также метод withFilter, ту же операцию можно выполнить с фильтрами-выражениями:

```
scala> val valid = for (res <- fut if res > 0)
yield res
valid: scala.concurrent.Future[Int] = ...
scala> valid.value
res2: Option[scala.util.Try[Int]] =
Some(Success(42))
```

```
scala> val invalid = for (res <- fut if res < 0)
yield res
invalid: scala.concurrent.Future[Int] = ...

scala> invalid.value
res3: Option[scala.util.Try[Int]] =
  Some(Failure(java.util.NoSuchElementException:
  Future.filter predicate is not satisfied))
```

Имеющийся в Future метод collect позволяет проверить на допустимость значение фьючерса и превратить все в одну операцию. Если переданная collect частично применяемая функция определена в результате фьючерса, то фьючерс, возвращаемый collect, будет успешно завершен с этим значением, преобразованным функцией:

```
scala> val valid =
fut collect { case res if res > 0 => res + 46 }
valid: scala.concurrent.Future[Int] = ...

scala> valid.value
res17:        Option[scala.util.Try[Int]]        =
Some(Success(88))
```

В противном случае этот фьючерс даст сбой с выдачей исключения NoSuchElementException:

```
scala> val invalid =
          fut collect { case res if res < 0 => res
+ 46 }
invalid: scala.concurrent.Future[Int] = ...

scala> invalid.value
```

```
res18: Option[scala.util.Try[Int]] =
  Some(Failure(java.util.NoSuchElementException:
   Future.collect partial function is not defined
at: 42))
```

Обработка сбоев: failed, fallBackTo, recover и recoverWith

Технология фьючерсов в Scala позволяет работать с теми фьючерсами, которые дали сбой, с помощью методов `failed`, `fallBackTo`, `recover` и `recoverWith`. Метод `failed` превратит давший сбой фьючерс любого типа в успешно завершенный объект типа `Future[Throwable]`, хранящий в себе исключение, ставшее причиной сбоя. Рассмотрим пример:

```
scala> val failure = Future { 42 / 0 }
failure: scala.concurrent.Future[Int] = ...

scala> failure.value
res23: Option[scala.util.Try[Int]] =
  Some(Failure(java.lang.ArithmeticException: / by
zero))
scala> val expectedFailure = failure.failed
expectedFailure:
scala.concurrent.Future[Throwable] = ...

scala> expectedFailure.value
res25: Option[scala.util.Try[Throwable]] =
  Some(Success(java.lang.ArithmeticException: / by
zero))
```

Если фьючерс, в отношении которого вызван метод `failed`, в конечном счете успешно завершится, фьючерс, возвращенный методом `failed`, сам даст сбой с выдачей исключения `NoSuchElementException`. Поэтому метод `failed` применим

только в том случае, когда ожидается сбой фьючерса. Рассмотрим пример:

```
scala> val success = Future { 42 / 1 }
success: scala.concurrent.Future[Int] = ...

scala> success.value
res21:        Option[scala.util.Try[Int]]        =
Some(Success(42))

scala> val unexpectedSuccess = success.failed
unexpectedSuccess:
scala.concurrent.Future[Throwable] = ...

scala> unexpectedSuccess.value
res26: Option[scala.util.Try[Throwable]] =
  Some(Failure(java.util.NoSuchElementException:
  Future.failed not completed with a throwable.))
```

Метод `fallbackTo` позволяет предоставить альтернативный фьючерс для использования в том случае, если фьючерс, в отношении которого был вызван `fallbackTo`, даст сбой. Рассмотрим пример, в котором давший сбой фьючерс уступает успешно завершаемому фьючерсу:

```
scala> val fallback = failure.fallbackTo(success)
fallback: scala.concurrent.Future[Int] = ...

scala> fallback.value
res27:        Option[scala.util.Try[Int]]        =
Some(Success(42))
```

Если исходный фьючерс, в отношении которого был вызван метод `fallbackTo`, даст сбой, то этот сбой, переданный

fallbackTo, по сути, игнорируется. Фьючерс, возвращенный методом fallbackTo, даст сбой с выдачей исходного исключения. Рассмотрим пример:

```
scala> val failedFallback = failure.fallbackTo(
          Future { val res = 42; require(res < 0);
res }
        )
failedFallback: scala.concurrent.Future[Int] = ...

scala> failedFallback.value
res28: Option[scala.util.Try[Int]] =
  Some(Failure(java.lang.ArithmeticException: / by
zero))
```

Метод recover позволяет преобразовать давший сбой фьючерс в успешно завершенный, разрешая результату успешно завершенного фьючерса пройти через него в неизмененном виде. Например, в отношении фьючерса, давшего сбой с выдачей исключения ArithmeticException, можно воспользоваться методом recover, чтобы превратить сбой в успешное завершение:

```
scala> val recovered = failedFallback recover {
         case ex: ArithmeticException => -1
       }
recovered: scala.concurrent.Future[Int] = ...

scala> recovered.value
res32:        Option[scala.util.Try[Int]]        =
Some(Success(-1))
```

Если исходный фьючерс не дал сбой, то фьючерс, возвращенный методом recover, завершится с тем же значением:

```
scala> val unrecovered = fallback recover {
         case ex: ArithmeticException => -1
       }
unrecovered: scala.concurrent.Future[Int] = ...

scala> unrecovered.value
res33:        Option[scala.util.Try[Int]]        =
Some(Success(42))
```

Аналогично этому, если частично применяемая функция, переданная методу, не определена при исключении, с которым в конечном счете дал сбой исходный фьючерс, то далее пройдет исходный отказ:

```
scala>   val   alsoUnrecovered   =   failedFallback
recover {
         case ex: IllegalArgumentException => -2
       }
alsoUnrecovered:   scala.concurrent.Future[Int]   =
...
scala> alsoUnrecovered.value

res34: Option[scala.util.Try[Int]] =
  Some(Failure(java.lang.ArithmeticException: / by
zero))
```

Метод `recoverWith` похож на метод `recover`, за исключением того, что вместо восстановления в виде значения, как в `recover`, метод `recoverWith` позволяет выполнять восстановление в значение фьючерса. Рассмотрим пример:

```
scala>   val   alsoRecovered   =   failedFallback
recoverWith {
          case ex: ArithmeticException => Future {
42 + 46 }
```

```
        }
alsoRecovered: scala.concurrent.Future[Int] = ...

scala> alsoRecovered.value
res35:        Option[scala.util.Try[Int]]        =
Some(Success(88))
```

Как и в случае применения `recover`, если либо исходный фьючерс не даст сбой, либо частично применяемая функция, переданная методу `recoverWith`, не будет определена при исключении, с которым в конечном счете дал сбой исходный фьючерс. Через фьючерс, возвращенный методом `recoverWith`, будет передан исходный успешный (или сбойный) результат.

Отображение обеих возможностей: transform

Определенный в `Future` метод `transform` получает две функции, с помощью которых он должен преобразовать фьючерс: одну для использования в случае успеха, другую — в случае сбоя:

```
scala> val first = success.transform(
         res => res * -1,
         ex => new Exception("see cause", ex)
       )
first: scala.concurrent.Future[Int] = ...
```

Если фьючерс завершается успешно, используется первая функция:

```
scala> first.value
res42:        Option[scala.util.Try[Int]]        =
Some(Success(-42))
```

Если фьючерс дает сбой, используется вторая функция:

```
scala> val second = failure.transform(
           res => res * -1,
           ex => new Exception("see cause", ex)
         )
second: scala.concurrent.Future[Int] = ...

scala> second.value
res43: Option[scala.util.Try[Int]] =
  Some(Failure(java.lang.Exception: see cause))
```

Заметьте, что с помощью показанного в предыдущих примерах метода `transform` превратить успешно завершенный фьючерс в сбойный и, наоборот, сбойный — в успешно завершенный невозможно. Чтобы упростить подобные преобразования, в Scala 2.12 была введена альтернативная перегружаемая форма `transform`, которая использует функцию из `Try` в `Try`. Рассмотрим несколько примеров:

```
scala> val firstCase = success.transform { // Scala 2.12
         case Success(res) => Success(res * -1)
         case Failure(ex) =>
           Failure(new Exception("see cause", ex))
       }
first: scala.concurrent.Future[Int] = ...

scala> firstCase.value
res6: Option[scala.util.Try[Int]] = Some(Success(-42))

scala> val secondCase = failure.transform {
         case Success(res) => Success(res * -1)
         case Failure(ex) =>
```

```
            Failure(new Exception("see cause", ex))
        }
secondCase: scala.concurrent.Future[Int] = ...

scala> secondCase.value
res8: Option[scala.util.Try[Int]] =
   Some(Failure(java.lang.Exception: see cause))
```

А вот как выглядит пример использования нового метода `transform` для превращения сбоя в успешное завершение:

```
scala> val nonNegative = failure.transform { // Scala 2.12
         case Success(res) => Success(res.abs + 1)
         case Failure(_) => Success(0)
       }
nonNegative: scala.concurrent.Future[Int] = ...

scala> nonNegative.value
res11: Option[scala.util.Try[Int]] = Some(Success(0))
```

Объединение фьючерсов: zip, Future.fold, Future.reduce, Future.sequence и Future.traverse

Класс `Future` и его объект-спутник предлагают методы объединения нескольких фьючерсов. Метод `zip` превратит два успешных фьючерса во фьючерс-кортеж из обоих значений. Рассмотрим пример:

```
scala> val zippedSuccess = success zip recovered
zippedSuccess: scala.concurrent.Future[(Int, Int)] = ...
```

```
scala> zippedSuccess.value
res46: Option[scala.util.Try[(Int, Int)]] =
    Some(Success((42,-1)))
```

Но если какой-либо из фьючерсов даст сбой, то фьючерс, возвращенный методом `zip`, также даст сбой с выдачей того же самого исключения:

```
scala> val zippedFailure = success zip failure
zippedFailure: scala.concurrent.Future[(Int, Int)]
= ...

scala> zippedFailure.value
res48: Option[scala.util.Try[(Int, Int)]] =
  Some(Failure(java.lang.ArithmeticException: / by
zero))
```

Если сбойными окажутся оба фьючерса, получающийся сбойный фьючерс будет содержать исключение, сохраненное в исходном фьючерсе — в том, в отношении которого был вызван метод `zip`.

Объект-спутник класса `Future` предлагает метод `fold`, позволяющий аккумулировать результат, получаемый из коллекции фьючерсов, имеющей тип `TraversableOnce`, выдавая результат фьючерса. Если успешными окажутся все фьючерсы коллекции, получающийся фьючерс будет успешным с аккумулированным результатом. Если любой из фьючерсов коллекции даст сбой, получающийся фьючерс будет сбойным. Если дадут сбой несколько фьючерсов, результат будет сбойным с тем самым исключением, с которым потерпел сбой первый фьючерс (самый первый в `TraversableOnce`-коллекции). Рассмотрим пример:

```
scala> val fortyTwo = Future { 21 + 21 }
```

```
fortyTwo: scala.concurrent.Future[Int] = ...
scala> val fortySix = Future { 23 + 23 }
fortySix: scala.concurrent.Future[Int] = ...

scala> val futureNums = List(fortyTwo, fortySix)
futureNums:  List[scala.concurrent.Future[Int]]  =
...

scala> val folded =
        Future.fold(futureNums)(0) { (acc, num) =>
          acc + num
        }
folded: scala.concurrent.Future[Int] = ...

scala> folded.value
res53:        Option[scala.util.Try[Int]]        =
Some(Success(88))
```

Метод `Future.reduce` выполняет свертку без нуля, используя в качестве стартового значения результат исходного фьючерса. Рассмотрим пример:

```
scala> val reduced =
        Future.reduce(futureNums) { (acc, num) =>
          acc + num
        }
reduced: scala.concurrent.Future[Int] = ...

scala> reduced.value
res54:        Option[scala.util.Try[Int]]        =
Some(Success(88))
```

Если передать `reduce` пустую коллекцию, получившийся фьючерс даст сбой с выдачей исключения

NoSuchElementException.

Метод Future.sequence выполняет преобразование TraversableOnce-коллекции фьючерсов во фьючерс, содержащий значения типа TraversableOnce. Например, в следующем примере sequence используется для преобразования List[Future[Int]] в Future[List[Int]]:

```
scala> val futureList = Future.sequence(futureNums)
futureList: scala.concurrent.Future[List[Int]] = ...

scala> futureList.value
res55: Option[scala.util.Try[List[Int]]] =
  Some(Success(List(42, 46)))
```

Метод Future.traverse превратит TraversableOnce-коллекцию из элементов любого типа в TraversableOnce-коллекцию фьючерсов и сведет ее во фьючерс значений TraversableOnce. Вот как, к примеру, Future.traverse превратит List[Int] в Future[List[Int]]:

```
scala> val traversed =
           Future.traverse(List(1, 2, 3)) { i => Future(i) }
traversed: scala.concurrent.Future[List[Int]] = ...

scala> traversed.value
res58: Option[scala.util.Try[List[Int]]] =
  Some(Success(List(1, 2, 3)))
```

Реализация побочных эффектов: foreach, onComplete и andThen

Иногда после завершения фьючерса требуется реализовать какой-то побочный эффект. Для этого в классе `Future` имеется несколько методов. Наиболее основательным из них является метод `foreach`, реализующий побочный эффект при успешном завершении фьючерса. Так, в следующем примере `println` выполняется только при успешном завершении фьючерса, но не в случае его сбоя:

```
scala> failure.foreach(ex => println(ex))

scala> success.foreach(res => println(res))
42
```

Поскольку выражение `for` без элемента `yield` будет перезаписано в вызов `foreach`, того же самого эффекта можно добиться и с его использованием:

```
scala> for (res <- failure) println(res)

scala> for (res <- success) println(res)
42
```

Класс `Future` также предоставляет два метода для регистрации функций обратного вызова. Метод `onComplete` будет выполнен независимо от успешного или сбойного завершения фьючерса. Функции будет передано значение типа `Try` либо в виде значения подтипа `Success`, содержащего результат успешно завершенного фьючерса, либо в виде значения подтипа `Failure`, содержащего исключение, ставшее причиной сбоя фьючерса. Рассмотрим пример:

```
scala> import scala.util.{Success, Failure}
import scala.util.{Success, Failure}
```

```
scala> success onComplete {
         case Success(res) => println(res)
         case Failure(ex) => println(ex)
       }
42

scala> failure onComplete {
         case Success(res) => println(res)
         case Failure(ex) => println(ex)
       }
java.lang.ArithmeticException: / by zero
```

Класс Future не дает гарантий того, что будет соблюден какой-то порядок выполнения функций обратного вызова, зарегистрированных с помощью onComplete. Если нужно обязать придерживаться какого-то порядка выполнения функций обратного вызова, следует вместо этого метода воспользоваться методом andThen. Он возвращает новый фьючерс, являющийся отражением исходного фьючерса (как успешного, так и сбойного), в отношении которого вызывался andThen, но не будет завершен, пока не будет полностью выполнена функция обратного вызова:

```
scala> val newFuture = success andThen {
         case Success(res) => println(res)
         case Failure(ex) => println(ex)
       }
42
newFuture: scala.concurrent.Future[Int] = ...

scala> newFuture.value
res76:         Option[scala.util.Try[Int]]         =
Some(Success(42))
```

Следует заметить, что, если функция обратного вызова,

переданная andThen, выдает при выполнении исключение, это исключение не будет распространено на следующие функции обратного вызова или отмечено посредством получающегося в результате фьючерса.

Другие методы, добавленные в версии 2.12: flatten, zipWith и transformWith. Метод flatten, добавленный в версии 2.12, преобразует Future-объект, вложенный в другой Future-объект, в Future-объект вложенного типа. Например, flatten может преобразовать Future[Future[Int]] в Future[Int]:

```
scala> val nestedFuture = Future { Future { 42 } }
nestedFuture: Future[Future[Int]] = ...

scala> val flattened = nestedFuture.flatten // Scala 2.12
flattened: scala.concurrent.Future[Int] = Future(Success(42))
```

Метод zipWith, добавленный в версии 2.12, по сути, пакует два Future-объекта вместе, затем применяет к получившемуся кортежу метод map. Рассмотрим процесс в два шага, в котором за zip следует map:

```
scala> val futNum = Future { 21 + 21 }
futNum: scala.concurrent.Future[Int] = ...

scala> val futStr = Future { "ans" + "wer" }
futStr: scala.concurrent.Future[String] = ...

scala> val zipped = futNum zip futStr
zipped: scala.concurrent.Future[(Int, String)] = ...

scala> val mapped = zipped map {
```

```
          case (num, str) => s"$num is the $str"
      }
mapped: scala.concurrent.Future[String] = ...

scala> mapped.value
res2: Option[scala.util.Try[String]] =
    Some(Success(42 is the answer))
```

Метод zipWith позволяет выполнять ту же операцию за один шаг:

```
scala> val fut = futNum.zipWith(futStr) { // Scala
2.12
          case (num, str) => s"$num is the $str"
      }
zipWithed: scala.concurrent.Future[String] = ...
scala> fut.value
res3: Option[scala.util.Try[String]] =
    Some(Success(42 is the answer))
```

Класс Future в Scala 2.12 также получил метод transformWith, который позволяет выполнить преобразование из Try в Future с использованием функции. Рассмотрим пример:

```
scala> val flipped = success.transformWith { //
Scala 2.12
          case Success(res) =>
                                Future { throw new
Exception(res.toString) }
          case Failure(ex) => Future { 21 + 21 }
      }
flipped: scala.concurrent.Future[Int] = ...

scala> flipped.value
```

```
res5: Option[scala.util.Try[Int]] =
    Some(Failure(java.lang.Exception: 42))
```

Метод `transformWith` похож на новый, добавленный в Scala 2.12 перегружаемый метод `transform`, за исключением того, что вместо выдачи `Try` переданной вами функции, как в `transform`, метод `transformWith` позволяет выдавать фьючерс.

32.4. Тестирование с применением Future-объектов

Одним из преимуществ фьючерсов Scala является то, что они помогают избегать блокировки. В большинстве реализаций JVM-машин после создания всего лишь нескольких тысяч потоков затратность контекстного переключения между потоками снизит производительность до неприемлемого уровня. Избегая блокировок, можно сохранить в рабочем состоянии намеченное конечное количество потоков. Тем не менее Scala позволяет при необходимости осуществлять блокировку на результате фьючерса. Блокировку в ожидании результата фьючерса обеспечивает имеющийся в Scala класс `Await`. Рассмотрим пример:

```
scala> import scala.concurrent.Await
import scala.concurrent.Await

scala> import scala.concurrent.duration._
import scala.concurrent.duration._

scala> val fut = Future { Thread.sleep(10000); 21
+ 21 }
fut: scala.concurrent.Future[Int] = ...

scala> val x = Await.result(fut, 15.seconds) //
Блокирует
x: Int = 42
```

Метод Await.result получает объект типа Future и объект типа Duration. Последний из них показывает продолжительность заданного Await.result ожидания завершения Future-объекта. В данном примере Duration-объект получил указание ждать 15 секунд. Следовательно, метод Await.result не достигнет превышения времени ожидания до завершения фьючерса с конечным результатом 42.

Одной из задач, где блокировка, несомненно, приветствуется, является тестирование асинхронного кода. После возвращения управления из Await.result появится возможность выполнить на основе результата вычисление, подобное утверждению при проведении теста:

```
scala> import org.scalatest.Matchers._
import org.scalatest.Matchers._

scala> x should be (42)
res0: org.scalatest.Assertion = Succeeded
```

В качестве альтернативного варианта можно воспользоваться конструкциями блокировки, предоставляемыми имеющимся в Scala трейтом ScalaFutures. К примеру, метод futureValue, подразумеваемо добавляемый ScalaFutures к Future-объекту, установит блокировку до завершения фьючерса. Если фьючерс завершится сбоем, метод futureValue выдаст исключение TestFailedException с описанием проблемы. Если фьючерс завершится успешно, методом futureValue будет возвращен успешный результат и в отношении него будет позволено использовать утверждение:

```
scala> import
org.scalatest.concurrent.ScalaFutures._
import org.scalatest.concurrent.ScalaFutures._
```

```
scala> val fut = Future { Thread.sleep(10000); 21
+ 21 }
fut: scala.concurrent.Future[Int] = ...

scala> fut.futureValue should be (42) // Блокирует
futureValue
res1: org.scalatest.Assertion = Succeeded
```

Хотя применение блокировки при тестировании зачастую приносит пользу, в ScalaTest 3.0 добавлены асинхронные стили тестирования, позволяющие тестировать фьючерсы без использования блокировки. Располагая фьючерсом, вместо использования блокировки и проверки утверждения в отношении результата можно отобразить утверждения непосредственно на этот фьючерс и возвратить получившийся `Future[Assertion]` среде ScalaTest. Соответствующий пример показан в листинге 32.1. Когда фьючерс-утверждение завершится, ScalaTest инициирует события (успешного завершения теста, его сбойного завершения и т. п.), чтобы отрапортовать о результате теста в асинхронном режиме.

Листинг 32.1. Возвращение в ScalaTest фьючерса-утверждения

```
import org.scalatest.AsyncFunSpec
import scala.concurrent.Future

class AddSpec extends AsyncFunSpec {
  def addSoon(addends: Int * ): Future[Int] =
        Future { addends.sum }
  describe("addSoon") {
     it("will eventually compute a sum of passed
Ints") {
      val futureSum: Future[Int] = addSoon(1, 2)
```

```
        // Утверждение можно отобразить на Future-
объект, затем возвратить
      // получившийся фьючерс Future[Assertion] по
адресу ScalaTest:
      futureSum map { sum => assert(sum == 3) }
    }
  }
}
```

Использование асинхронного тестирования иллюстрирует общий принцип работы с фьючерсами: находясь в пространстве фьючерсов, старайтесь не покидать этого пространства. Не ставьте блокировку на фьючерсе с последующим вычислением результата. Продолжайте работу в асинхронном режиме, выполняя серии преобразований, каждое из которых возвращает для преобразования новый фьючерс. Для получения результатов в пространстве фьючерсов регистрируйте по завершении фьючерсов побочные эффекты, выполняемые асинхронно. Такой подход поможет вам максимально задействовать все имеющиеся потоки.

Резюме

Программирование многопоточных вычислений наделяет вас большими возможностями. Оно позволяет упростить код и получить преимущества от наличия в системе нескольких процессоров. К великому сожалению, большинство широко используемых многопоточных примитивов, потоков, блокировок и мониторов похоже на минное поле, усыпанное взаимными блокировками и соперничеством потоков. Фьючерсы обеспечивают проход в этом минном поле, позволяющий создавать многопоточные программы без особого риска столкновения со взаимными блокировками и условиями соперничества потоков. В этой главе среди всего прочего были представлены некоторые основные конструкции для работы с фьючерсами в Scala, включая

способы создания фьючерсов, приемы их преобразования и способы тестирования. Затем было показано, как можно воспользоваться этими конструкциями в качестве части общего стиля работы с фьючерсами.

149 Что касается Scala.js, глобальный контекст выполнения помещает задачи в очередь событий JavaScript.

150 Следует заметить, что у Java Future также имеется способ обработки потенциальной возможности выдачи исключения в ходе асинхронного вычисления: имеющийся в данной технологии метод get выдаст это исключение заключенным в ExecutionException.

151 Выражение for, показанное в этом примере, будет переписано в виде вызова fut1.flatMap, переданного в функцию, которая вызывает fut2.map : fut1.flatMap(x => fut2.map(y => x + y)).

33. Синтаксический анализ с применением комбинаторов

Временами могут возникать потребности в обработке данных на небольшом языке специального назначения. Например, потребуется прочитать конфигурационные файлы вашей программы и сделать их более подходящими для внесения изменений вручную, чем файлы формата XML. Или же в вашей программе понадобится поддержка языка ввода данных, например при поиске информации с использованием булевых операторов («Компьютер, найди мне фильм с космическими кораблями и без любовных историй»). Независимо от причин, вам нужен синтаксический анализатор, или *парсер*. Требуется способ превращения данных на языке их ввода в некую структуру данных, которую уже можно будет обработать вашей программой.

Выбор, по сути, невелик. Есть вариант создания собственного парсера (и лексического анализатора). Если вы не узкий специалист в этом деле, то справиться с подобной задачей будет нелегко. И даже если квалификация позволяет, на это все равно уйдет много времени.

Альтернативным вариантом будет использование парсер-генератора. Таких генераторов немного. Наиболее известные — Yacc и Bison, которые применяются для парсеров, создаваемых на языке C, и ANTLR, применяемый для парсеров, создаваемых на языке Java. Скорее всего, для работы вам понадобится также сканер-генератор из разряда Lex, Flex или JFlex. Если не принимать во внимание некоторые неудобства, такое решение может стать наилучшим. Для этого понадобится изучить новые программные средства, включая их иногда малопонятные сообщения об ошибках. Также нужно будет понять, как связать выходные данные этих средств со своей программой. Все это может ограничить выбор языка программирования и усложнить цепочку

применяемых инструментальных средств.

В этой главе представлен третий вариант: вместо использования внешнего предметно-ориентированного языка парсер-генератора воспользоваться *внутренним предметно-ориентированным языком* (domain specific language), или, для краткости, внутренним DSL. Этот язык будет состоять из библиотеки *парсер-комбинаторов*, то есть функций и операторов, определенных в Scala и применяемых в качестве строительных блоков парсеров. Эти блоки будут поэлементно отображаться на грамматические конструкции произвольного содержания, облегчая их понимание.

В этой главе, в разделе 33.6, вводится только одна ранее не встречавшаяся особенность языка — использование псевдонимов с применением ключевого слова `this`. Однако весьма интенсивно используется ряд других особенностей языка, рассмотренных в предыдущих главах. Среди них немаловажную роль играют параметризированные типы, абстрактные типы, применение функций в качестве объектов, перегрузка операторов, использование параметров до востребования и подразумеваемых преобразований. В этой главе будет показано, как эти элементы языка могут быть скомбинированы при создании библиотеки весьма высокого уровня.

Рассматриваемые в главе понятия будут, наверное, более высокого порядка, чем те, что изучались прежде. Эта глава будет полезна тем, у кого есть неплохие базовые знания об устройстве компилятора, поскольку она поможет расширить кругозор. Но для усвоения изложенного здесь материала нужно неплохо разбираться в обычной и бесконтекстной грамматике. Если же эта область вызывает затруднения, материал данной главы можно свободно пропустить.

33.1. Пример: арифметические выражения

Начнем с примера. Предположим, что нужно сконструировать парсер для арифметических выражений, состоящих из чисел с плавающей точкой, круглых скобок и бинарных операторов +, -, * и /. В качестве первого шага всегда нужно записывать грамматику анализируемого парсером языка. Грамматика для арифметических выражений имеет следующий вид:

```
        expr ::= term { "+" term | "-" term}.
           term ::= factor { " * " factor | "/"
factor}.
     factor ::= floatingPointNumber | "(" expr ")"
.
```

Здесь знак | обозначает альтернативный набор правил, а знаки { ... } — повторение (нуль и более раз). И хотя в данном примере знаки [...] не используются, они обозначают необязательное наличие.

Этой бесконтекстной грамматикой формально определяется язык арифметических выражений. Каждое выражение, представленное в виде expr, является синтаксическим термом (term), за которым могут располагаться последовательность операторов + или - и дополнительные термы. Терм является фактором (factor), за которым, возможно, идут последовательность операторов * или / и дополнительные факторы. Фактор является либо числовым литералом, либо выражением в круглых скобках. Заметьте, что в грамматике уже закодирована относительная степень приоритетности операторов. Например, у оператора * привязка более крепкая, чем у +, поскольку операция * дает терм, а операция + дает выражение, а оно может содержать термы, но терм не может содержать выражение, если только оно не заключено в круглые скобки.

Итак, грамматика определена. Что делать дальше? Если используются имеющиеся в Scala парсер-комбинаторы, то основное уже сделано! Нужно лишь выполнить некоторые

систематизированные текстовые замены и, как показано в листинге 33.1, заключить парсер в класс.

Листинг 33.1. Парсер арифметического выражения

```
import scala.util.parsing.combinator._

class Arith extends JavaTokenParsers {
   def expr: Parser[Any] = term~rep("+"~term | "-
"~term)
   def term: Parser[Any] = factor~rep(" * "~factor
| "/"~factor)
   def factor: Parser[Any] = floatingPointNumber |
"("~expr~")"
}
```

Парсеры для арифметических выражений содержатся в классе, являющемся наследником трейта `JavaTokenParsers`. Этот трейт предоставляет основной механизм для написания парсера, а также некоторые элементарные парсеры, распознающие некоторые классы слов: идентификаторы, строковые литералы и числа. В примере, показанном в листинге 33.1, нужен всего лишь элементарный парсер `floatingPointNumber`, наследуемый из этого трейта.

Набор правил для арифметических выражений представлен тремя определениями в классе `Arith`. Вы увидите, что они весьма точно соответствуют набору правил бесконтекстной грамматики. Фактически эту часть можно создать из бесконтекстной грамматики автоматически, выполняя ряд простых замен текста.

Каждое правило становится методом, поэтому перед ним нужно ставить префикс `def`.

37. Типом результата любого метода является `Parser[Any]`,

следовательно, нужно заменить обозначение `::=` кодом `: Parser[Any] =`. Значение типа `Parser[Any]` и порядок его уточнения будут разъяснены в этой главе чуть позже.

38. В грамматике последовательная композиция подразумевалась, но в программе она выражена явным оператором `~`. Поэтому нужно вставить оператор `~` между двумя любыми последовательными обозначениями правила. В примере, показанном в листинге 33.1, мы решили не ставить пробелы с обеих сторон от оператора `~`, чтобы приблизить код к визуальному отображению грамматики, а просто заменили пробелы символами `~`.

39. Повторения выражены кодом `rep( ... )`, заменившим форму `{ ... }`. Аналогично этому (хотя в примере это не показано) необязательные элементы выражаются кодом `opt( ... )`, заменяющим форму `[ ... ]`.

40. Точка (`.`) в конце всех правил опущена, но при желании вместо нее можно поставить точку с запятой (`;`).

Вот, собственно, и все. В получившемся в итоге классе `Arith` определяются три парсера, `expr`, `term` и `factor`, которые могут использоваться для синтаксического анализа арифметических выражений и их частей.

33.2. Запуск парсера

Парсер можно использовать со следующей небольшой программой:

```
object ParseExpr extends Arith {
  def main(args: Array[String]) = {
    println("input : " + args(0))
    println(parseAll(expr, args(0)))
```

```
  }
}
```

Объект `ParseExpr` определяет основной метод, выполняющий парсинг переданного ему первого аргумента командной строки. Им выводятся исходный аргумент ввода, а затем его проанализированная парсером версия. Парсинг выполняется следующим выражением:

```
parseAll(expr, input)
```

В нем парсер `expr` применяется к указанным входным данным `input`. Ожидается соответствие всем входным данным, то есть предполагается, что за подвергающимся парсингу выражением больше нет никаких символов. Есть также метод `parse`, позволяющий анализировать префикс введенных данных, оставляя все остальное непрочитанным.

Арифметический парсер можно запустить с помощью следующей команды:

```
$ scala ParseExpr "2 * (3 + 7)"
input: 2 * (3 + 7)
[1.12] parsed: ((2~List((*~((((~((3~List())~List((+
~(7~List())))))~)))))~List())
```

Если судить на основании выходных данных, пасер успешно проанализировал строку ввода до позиции `[1.12]`. Это означает первую строку и двенадцатый столбец, иными словами, парсером обработана вся введенная строка. Результат, показанный после `parsed:`, мы пока проигнорируем. Пользы от него немного, а как получить от парсера более конкретные результаты, вы поймете чуть позже.

Можно также попробовать ввести строку ввода с недопустимым выражением. Например, поставить одну лишнюю закрывающую скобку:

```
$ scala ParseExpr "2 * (3 + 7))"
input: 2 * (3 + 7))
[1.12] failure: '-' expected but ')' found

2 * (3 + 7))
           ^
```

В данном случае парсер `expr` обработает все вплоть до последней закрывающей скобки, которая не формирует часть арифметического выражения. Затем метод `parseAll` выдаст сообщение об ошибке, в котором будет сказано, что на месте закрывающей скобки ожидался оператор `-`. Чуть позже в этой главе будет разъяснено, почему было выдано именно такое сообщение об ошибке и как можно исправить ситуацию.

33.3. Основные парсеры — регулярные выражения

В парсере арифметических выражений используется еще один парсер по имени `floatingPointNumber`. Этот парсер, унаследованный от родительского, по отношению к `Arith` трейта `JavaTokenParsers` распознает число с плавающей точкой в формате Java. А как поступить, когда нужно выполнить парсинг чисел в формате, несколько отличающемся от формата, используемого в Java? В такой ситуации можно воспользоваться *парсером — регулярным выражением*.

Замысел заключается в том, что в качестве парсера можно использовать любое регулярное выражение. Это выражение выполняет парсинг всех строк, которым оно может соответствовать. В результате получается обработанная парсингом строка. Например, парсер — регулярное выражение, показанный в листинге 33.2, дает описание идентификаторов Java.

Листинг 33.2. Парсер — регулярное выражение для идентификаторов Java

```
object MyParsers extends RegexParsers {
   val ident: Parser[String] = """[a-zA-Z_]\w *
""".r }
```

Объект MyParsers в листинге 33.2 является наследником трейта RegexParsers, в то время как Arith — наследник JavaTokenParsers. Имеющиеся в Scala комбинаторы парсинга располагаются в иерархии трейтов, целиком содержащейся в пакете scala.util.parsing.combinator. Трейтом на самом верхнем уровне является Parsers, и в нем определяется самая общая среда парсинга для любого вида входных данных. На один уровень ниже располагается трейт RegexParsers, требующий, чтобы входные данные представляли собой последовательность символов, и предоставляемый для парсинга с применением регулярных выражений. Еще более специализированным является трейт JavaTokenParsers, реализующий парсеры для основных классов слов (или токенов) в соответствии с определением, используемым в Java.

33.4. Еще один пример: JSON

JSON (JavaScript Object Notation) является популярным форматом обмена данными. Рассмотрим процесс создания парсера для этого формата. Грамматика, дающая описание синтаксиса JSON, имеет следующий вид:

```
value ::= obj | arr | stringLiteral |
          floatingPointNumber |
          "null" | "true" | "false" .
  obj ::= "{" [members] "}" .
  arr ::= "[" [values] "]" .
```

```
members ::= member { "," member}.
 member ::= stringLiteral ":" value.
 values ::= value { "," value}.
```

JSON-значение является объектом, массивом, строкой, числом или одним из трех зарезервированных слов: `null`, `true` или `false`. JSON-объект является последовательностью (возможно, пустой) элементов, отделенных друг от друга запятыми и заключенных в круглые скобки. Каждый элемент является парой «строка — значение», где строка и значение отделены друг от друга двоеточием. И наконец, JSON-массив является последовательностью значений, отделенных друг от друга запятыми и заключенных в квадратные скобки. В качестве примера в листинге 33.3 содержится адресная книга, форматированная в виде JSON-объекта.

Листинг 33.3. Данные в JSON-формате

```
{
  "address book": {
    "name": "John Smith",
    "address": {
       "street": "10 Market Street",
       "city" : "San Francisco, CA",
       "zip" : 94111
    },
    "phone numbers": [
      "408 338-4238",
      "408 111-6892"
    ]
  }
}
```

При использовании имеющихся в Scala парсер-комбинаторов выполнить синтаксический анализ подобных данных не составляет особого труда. Полнофункциональный парсер показан в листинге 33.4. Он имеет ту же структуру, что и парсер арифметических выражений. Здесь снова применяется прямое отображение правил на грамматику JSON. В наборе правил используется одно сокращение, упрощающее грамматику: комбинатор `repsep` выполняет парсинг последовательности термов (возможно, пустой), отделенных друг от друга заданной строкой разделителя. Например, в коде, показанном в листинге 33.4, `repsep(member, ",")` выполняет парсинг последовательности, элементы которой, представленные термами, отделены друг от друга запятыми. В остальных случаях набор правил в парсере в точности соответствует набору правил в грамматике, как это было в случае с парсерами для арифметических выражений.

Листинг 33.4. Простой парсер формата JSON

```
import scala.util.parsing.combinator._
class JSON extends JavaTokenParsers {
  def value : Parser[Any] = obj | arr |
                            stringLiteral |
                            floatingPointNumber |
                                 "null" | "true" |
"false"
   def obj    : Parser[Any] = "{"~repsep(member,
","}"
   def arr    : Parser[Any] = "["~repsep(value,
","}"]"
           def     member:     Parser[Any]     =
stringLiteral~":"~value
}
```

Чтобы проверить JSON-парсеры в работе, немного изменим среду, чтобы парсер работал с файлом, а не с командной строкой:

```
import java.io.FileReader
object ParseJSON extends JSON {
  def main(args: Array[String]) = {
    val reader = new FileReader(args(0))
    println(parseAll(value, reader))
  }
}
```

Метод `main` в этой программе сначала создает объект считывания файла — `FileReader`. Затем анализирует символы, возвращенные этим считывателем в соответствии со значением правила грамматики JSON. Следует заметить, что `parseAll` и `parse` существуют в перегружаемых вариантах: оба этих метода могут получать последовательность символов или дополнительно, в качестве второго аргумента, считыватель ввода.

Если объект «адресная книга», показанный в листинге 33.3, сохранить в файле с именем **address-book.json** и запустить в отношении этого файла программу `ParseJSON`, будет получен следующий результат:

```
$ scala ParseJSON address-book.json
[13.4] parsed: (({~List((("address book"~:)~(({~List(((
"name"~:)~"John Smith"), (("address"~:)~(({~List(((
"street"~:)~"10 Market Street"), (("city"~:)~"San Francisco
,CA"), (("zip"~:)~94111)))~})), (("phone numbers"~:)~(([~
List("408 338-4238", "408 111-6892"))~])))~})))~})
```

33.5. Вывод парсера

Программа `ParseJSON` успешно справляется с парсингом адресной книги в формате JSON. Но вывод парсера все же выглядит странно. Он похож на последовательность, составленную из обрывков, склеенных вместе с помощью комбинаций из списков входных данных и символов `~`. Пользы от него немного. Читателю с ним труднее разобраться, чем с входными данными, к тому же он слишком слабо организован, чтобы его легко проанализировал компьютер. Пора с этим что-нибудь сделать.

Чтобы понять, что нужно сделать, сначала следует узнать, что именно в среде комбинаторов возвращают в качестве результата отдельно взятые парсеры (в том случае, если они успешно выполнят парсинг входных данных). Здесь действуют следующие правила.

Каждый парсер, записанный как строка (такая как `"{"`, или `":"`, или `"null"`), сам возвращает проанализированную строку.

41. Парсер — регулярное выражение, такой как `"""[a-zA-Z_]\w*"""".r`, также сам возвращает проанализированную строку. То же самое справедливо и для таких парсеров — регулярных выражений, как `stringLiteral` или `floatingPointNumber`, наследуемых из трейта `JavaTokenParsers`.

42. Последовательная композиция `P~Q` возвращает результат как `P`, так и `Q`. Эти результаты возвращаются в экземпляре case-класса, который также записывает `~`. Следовательно, если `P` возвращает `"true"`, а `Q` возвращает `"?"`, то последовательная композиция `P~Q` возвращает `~("true", "?")`, что выводится как `(true~?)`.

43. Альтернативная композиция `P | Q` возвращает результат либо `P`, либо `Q` в зависимости от того, какой из этих парсеров успешно завершит свою работу.

44. Повторение `rep(P)` или `repsep(P, separator)` возвращает список результатов всех запусков `P`.

45. Вариант `opt(P)` возвращает экземпляр имеющегося в Scala типа `Option`. Он возвращает `Some(R)`, если работа `P` завершится успешно с результатом `R`, и `None`, если `P` даст сбой.

Теперь, руководствуясь этими правилами, можно понять, почему вывод парсера появляется именно в том виде, который был показан в предыдущих примерах. Но более удобным его восприятие от этого не стало. Намного лучше было бы отобразить JSON-объект на внутреннее представление Scala, изображающее смысл JSON-значения. Следующее представление было бы более естественным.

- JSON-объект представляется как Scala-отображение типа `Map[String, Any]`. Каждый элемент представлен связанной в представлении парой «ключ — значение».
- JSON-массив представляется как список Scala типа `List[Any]`.
- JSON-строка представляется как Scala `String`.
- Числовой литерал JSON представляется как Scala `Double`.
- Значения `true`, `false` и `null` представляются как значения Scala с теми же именами.

Чтобы выдать такое представление, нужно воспользоваться еще одной формой комбинации для парсеров — `€€`.

Оператор `€€` выполняет преобразования результата парсера. Выражения, использующие этот оператор, имеют вид `P €€ f`, где `P` обозначает парсер, а `f` — функцию. `P €€ f` анализирует предложения так же, как просто `P`. Когда `P` возвращает некий

результат R, результатом P €€ f становится f(R).

Рассмотрим в качестве примера парсер, анализирующий число с плавающей точкой и превращающий его в значение Scala типа Double:

```
floatingPointNumber ^^ (_.toDouble)
```

А так выглядит парсер, анализирующий строку "true" и возвращающий булево Scala-значение true:

```
"true" ^^ (x => true)
```

Теперь перейдем к более сложным преобразованиям. Вот как выглядит новая версия парсера для JSON-объектов, возвращающих Scala Map:

```
def obj: Parser[Map[String, Any]] = // Этот код
можно улучшить
  "{"~repsep(member, ","~"}" ^^
    { case "{"~ms~"}" => Map() ++ ms }
```

Вспомним, что оператор ~ выдает в качестве результата экземпляр класса с таким же именем — ~. А так выглядит определение этого класса, являющегося внутренним классом трейта Parsers:

```
case class ~ [+A, +B](x: A, y: B) {
  override def toString = "(" + x + "~" + y + ")"
}
```

Одинаковые имена для класса и метода — комбинатора последовательностей ~ выбраны не случайно. Это позволяет сопоставлять результаты парсера с шаблонами, имеющими такую же структуру, что и сами парсеры. Например, шаблон "{"~ms~"}" соответствует строке результата "{", за которой следует

переменная результата ms, а завершающей является строка результата "}". Этот шаблон в точности соответствует тому, что возвращает парсер слева от €€.

В его более открытых версиях, где сначала идет оператор ~, тот же шаблон считывает ~(~("{", ms), "}"), но выглядит это менее понятно.

Шаблон "{"~ms~"}" нужен для избавления от фигурных скобок, чтобы можно было получить список элементов, являющийся результатом работы парсера repsep(member, ","). В подобных случаях имеется альтернатива, позволяющая избегать выдачи ненужных результатов работы парсеров, которые немедленно отбрасываются поиском по шаблону. В этом альтернативном варианте используются парсер-комбинаторы ~> и <~. Они обозначают последовательную композицию, подобную ~, но ~> сохраняет только результаты своего правого операнда, а <~ — только результаты левого операнда. При использовании этих комбинаторов парсер JSON-объекта может быть выражен более кратко:

```
def obj: Parser[Map[String, Any]] =

"{"~> repsep(member, ",") <~"}" ^^ (Map() ++ _)
```

Полноценный парсер JSON-данных, возвращающий вполне выразительные результаты, показан в листинге 33.5.

Листинг 33.5. Полноценный JSON-парсер, возвращающий значимые результаты

```
import scala.util.parsing.combinator._

class JSON1 extends JavaTokenParsers {
```

```
  def obj: Parser[Map[String, Any]] =
     "{"~> repsep(member, ",") <~"}" ^^ (Map() ++
_)

  def arr: Parser[List[Any]] =
    "["~> repsep(value, ",") <~"]"

  def member: Parser[(String, Any)] =
    stringLiteral~":"~value ^^
      { case name~":"~value => (name, value) }
  def value: Parser[Any] = (
    obj
  | arr
  | stringLiteral
  | floatingPointNumber ^^ (_.toDouble)
  | "null"  ^^ (x => null)
  | "true"  ^^ (x => true)
  | "false" ^^ (x => false)
  )
}
```

Если запустить этот парсер в отношении файла **address-book.json**, будет (после добавления ряда символов новой строки и отступов) получен следующий результат:

```
$ scala JSON1Test address-book.json
[14.1] parsed: Map(
  address book -> Map(
    name -> John Smith,
    address -> Map(
      street -> 10 Market Street,
      city -> San Francisco, CA,
      zip -> 94111),
```

```
      phone numbers -> List(408 338-4238, 408 111-
6892)
   )
)
```

И это, собственно, все, что нужно знать, чтобы приступить к написанию собственных парсеров. В качестве шпаргалки в табл. 33.1 перечислены все рассмотренные до сих пор парсер-комбинаторы.

Таблица 33.1. Сводка парсер-комбинаторов

Парсер-комбинатор	Определение
"..."	Литерал
"...".r	Регулярное выражение
P~Q	Последовательная композиция
P <~ Q, P ~> Q	Последовательная композиция; сохраняет только левую или правую часть
P | Q	Альтернативная композиция
opt(P)	Возвращение Option-значения
rep(P)	Повторение
repsep(P, Q)	Чередующееся повторение
P €€ f	Получающееся преобразование

Сравнение символьных и буквенно-цифровых имен. Для многих парсер-комбинаторов, указанных в табл. 33.1, используются символьные имена. У такого подхода есть как преимущества, так и недостатки. К последним относится то, что символьные имена нужно запоминать. Пользователи, не знакомые со Scala-библиотеками парсинга с применением комбинаторов, скорое всего, не поймут значения ~, ~> или €€. К положительной стороне можно отнести краткость символьных имен и возможность правильно выбрать их принадлежность к той или

иной степени приоритетности и ассоциативности. Например, парсер-комбинаторы ~, €€ и | выбраны преднамеренно в порядке убывания степени приоритетности. Обычное грамматическое правило состоит из альтернатив, имеющих парсинг-часть и часть преобразования. Парсинг-часть обычно содержит несколько последовательных понятий, разделенных операторами ~. С выбранными степенями приоритетности ~, €€ и | можно записать такое грамматическое правило, не нуждаясь в расстановке круглых скобок.

Отключение подразумеваемого присутствия точек с запятой

Заметьте, что тело парсера значений в листинге 33.5 заключено в круглые скобки. Этот прием позволяет отключить подразумеваемое присутствие точек с запятой в выражениях парсеров. В разделе 4.2 было показано, что Scala подразумевает наличие между двумя строками кода точки с запятой, которая синтаксически может выступать в качестве разделителя инструкций, если только первая строка не заканчивается инфиксным оператором или две строки не заключены в круглые или квадратные скобки. Теперь можно ставить оператор | в конце каждого альтернативного варианта, а не начинать вариант с него:

```
def value: Parser[Any] =
  obj |
  arr |
  stringLiteral |
  ...
```

В этом случае не понадобятся круглые скобки вокруг тела парсера значений. Но некоторые предпочитают видеть оператор | в начале второго альтернативного варианта, а не в конце первого. Обычно

это приводит к нежелательной вставке между двумя строками точки с запятой:

```
  obj; // здесь подразумеваемо ставится точка с
запятой
| arr
```

Точка с запятой изменяет структуру кода, вызывая сбой компиляции. Заключение всего выражения в круглые скобки позволяет убрать точку с запятой и добиться правильной компиляции кода.

Более того, символьные операторы визуально занимают меньше места, чем буквенно-цифровые. Для парсера это важно, поскольку позволяет сконцентрироваться на исходной грамматике, а не на самих комбинаторах. Чтобы увидеть разницу, представьте на минуту, что последовательная композиция (~) была названа andThen, а альтернативный вариант (|) — rElse. Тогда парсер арифметического выражения, показанный в листинге 33.1, приобрел бы следующий вид:

```
class ArithHypothetical extends JavaTokenParsers {
  def expr: Parser[Any] =
    term andThen rep(("+" andThen term) orElse
                     ("-" andThen term))
  def term: Parser[Any] =
      factor andThen rep((" * " andThen factor)
orElse
                         ("/" andThen factor))
  def factor: Parser[Any] =
    floatingPointNumber orElse
    ("(" andThen expr andThen ")")
}
```

Как видите, код стал намного длиннее и в нем труднее «заметить» грамматику среди всех этих операторов и круглых скобок. С другой стороны, новичкам парсинга с использованием комбинаторов, наверное, будет проще разобраться в том, что именно делается с применением этого кода.

33.6. Реализация парсер-комбинаторов

В предыдущем разделе было показано, что имеющиеся в Scala парсер-комбинаторы предоставляют весьма удобные средства для создания ваших собственных парсеров. Представляя собой не что иное, как библиотеку Scala, они хорошо вписываются в ваши программы на Scala. Таким образом, очень легко объединить парсер с кодом, обрабатывающим поставляемые им результаты, или же выстроить парсер так, чтобы он получал свои данные из какого-то определенного источника (скажем, из файла, строки или массива символов).

Каким образом это достигается? Далее в главе вы сможете «заглянуть под капот» библиотеки парсер-комбинаторов. Вы увидите, что представляет собой парсер и как реализуются уже встречавшиеся элементарные парсеры и парсер-комбинаторы. Если ваши желания ограничиваются написанием простых парсер-комбинаторов, то эту часть главы можно спокойно пропустить. Однако, дочитав главу до конца, можно получить более глубокое представление о парсерах-комбинаторах в частности и о принципах разработки использующего комбинаторы предметно-ориентированного языка в общем.

Выбор между символьными и буквенно-цифровыми именами

В качестве руководства по выбору между символьными и буквенно-цифровыми именами мы даем следующие рекомендации.

Используйте символьные имена в тех случаях, когда у них уже имеется устоявшееся универсальное значение. Например, никто не станет рекомендовать для сложения чисел запись вида `add` вместо `+`. В ином случае отдавайте предпочтение алфавитно-цифровым именам, если хотите, чтобы ваш код был понятен случайным читателям.

Для предметно-ориентированных библиотек можно выбирать символьные имена, если они дают явное преимущество в разборчивости и не ожидается, что случайный читатель без основательной подготовки в предметной области сможет с ходу разобраться в коде.

Если искомые парсер-комбинаторы находятся в предметно-ориентированном языке более высокого уровня, у случайных читателей могут возникнуть проблемы с пониманием даже при использовании алфавитно-цифровых имен. Кроме того, специалистам символьные имена предоставляют явные преимущества, заключающиеся в разборчивости кода. Поэтому их использование в этом приложении мы считаем вполне оправданным.

Ядро имеющейся в Scala среды парсер-комбинаторов содержится в трейте `scala.util.parsing.combinator.Parsers`. Этот трейт определяется в типе `Parser`, как и все элементарные комбинаторы. Кроме тех мест, где это специально оговаривается, определения, рассматриваемые в следующих двух подразделах, находятся в этом трейте. Иначе говоря, подразумевается, что они содержатся в определении трейта, которое начинается со следующего кода:

```
package scala.util.parsing.combinator
```

```
trait Parsers {
  ... // код находится здесь, если не указано иное
}
```

По сути, `Parser` является функцией от некоторого типа входных данных к результату синтаксического анализа. В первом приближении тип может быть записан следующим образом:

```
type Parser[T] = Input => ParseResult[T]
```

Входные данные парсера

Иногда парсер считывает поток токенов, а не простую последовательность символов. Тогда для преобразования потока обычных символов в поток токенов используется отдельный лексический анализатор. Тип входных данных парсера определяется следующим образом:

```
type Input = Reader[Elem]
```

Класс `Reader` берется из пакета `scala.util.parsing.input`. Он похож на класс `Stream`, но вдобавок ко всему отслеживает позиции всех считываемых элементов. Отдельные входные элементы представлены типом `Elem`. Это элемент абстрактного типа трейта `Parsers`:

```
type Elem
```

Это означает, что подклассам и подтрейтам `Parsers` необходимо создавать экземпляр класса `Elem` для типа входных анализируемых парсером элементов. Например, `RegexParsers` и `JavaTokenParsers` определяют `Elem` эквивалентом `Char`. Но можно было бы также установить для `Elem` какой-либо другой тип, а именно тип токенов, возвращаемых отдельным лексическим анализатором (лексером).

Результаты парсера

Парсер может завершить работу с какими-либо входными данными либо успешно, либо со сбоем. Следовательно, для представления успеха и сбоя у класса `ParseResult` есть два подкласса:

```
sealed abstract class ParseResult[+T]
case class Success[T](result: T, in: Input)
  extends ParseResult[T]
case class Failure(msg: String, in: Input)
  extends ParseResult[Nothing]
```

Case-класс `Success` переносит результат, возвращенный парсером, в свой параметр `result`. Тип у результатов парсера произвольный, именно поэтому и `ParseResult`, и `Success`, и `Parser` параметризированы с параметром типа `T`. Параметр типа представляет разновидность результата, возвращенного данным парсером. `Success` получает также второй параметр `in`, ссылающийся на ту часть входных данных, которая следует сразу же за частью, поглощенной парсером. Это поле необходимо для парсеров, выстроенных в цепочку, чтобы один из них мог работать после другого. Заметьте, что это чисто функциональный подход к парсингу. Входные данные не считываются в виде побочного эффекта, но при этом они сохраняются в потоке. Парсер анализирует некоторую часть потока входных данных, после чего возвращает оставшуюся часть в своем результате.

Еще одним подклассом `ParseResult` является `Failure`. Этот класс получает в качестве параметра сообщение, где указана причина сбоя. Как и `Success`, класс `Failure` получает в качестве второго параметра оставшуюся часть потока входных данных. Это нужно не для цепочки парсеров (парсер не станет продолжать работу после сбоя), а для помещения сведений о месте сбоя в потоке входных данных в сообщение об ошибке. Заметьте, что результаты парсера определены в параметре `T` как ковариантные.

То есть парсер, возвращающий, к примеру, в качестве результата значения типа `String`, совместим с парсером, возвращающим значения типа `AnyRef`.

Класс Parser

Предыдущая характеристика парсеров как функций от входных данных к результатам парсера давала слишком упрощенное представление о них. Рассмотренные примеры показывают, что в парсерах реализуются также *методы* `~` для последовательной композиции двух парсеров и `|` — для их альтернативной композиции. Следовательно, на самом деле `Parser` является классом, наследуемым от функционального типа `Input => ParseResult[T]` и дополнительно определяющим эти методы:

```
abstract class Parser[+T] extends (Input =>
ParseResult[T])
{ p =>
  // произвольный метод, определяющий
  // поведение парсера.
  def apply(in: Input): ParseResult[T]
  def ~ ...
  def | ...
  ...
}
```

Поскольку парсеры являются функциями (то есть наследуются от них), им необходимо определение метода `apply`. Абстрактный метод `apply` можно увидеть в классе `Parser`, но он предназначен только для документирования, как и любой такой метод, в любом случае наследуемый из родительского типа `Input => ParseResult[T]` (вспомним, что этот тип является сокращенной формой записи от `scala.Function1[Input, ParseResult[T]]`). Метод `apply` все же нуждается в реализации в

отдельно взятых парсерах, являющихся наследниками абстрактного класса `Parser`. Эти парсеры будут рассмотрены после следующего раздела, посвященного использованию псевдонимов, создаваемых с помощью ключевого слова `this`.

Использование псевдонимов, создаваемых с помощью ключевого слова this

Тело класса `Parser` начинается с весьма странного выражения

```
abstract class Parser[+T] extends ... { p =>
```

Таким спецификатором, как `id =>`, стоящим сразу же после открывающей фигурной скобки шаблона класса, идентификатор `id` определяется в качестве псевдонима для `this` в данном классе. Это равнозначно записи:

```
val id = this
```

в теле класса, за исключением того, что компилятор Scala знает, что `id` является псевдонимом для `this`. Например, можно обратиться к закрытому элементу объекта `m` этого класса, воспользовавшись либо `id.m`, либо `this.m` — эти обращения абсолютно эквивалентны. Первое выражение не откомпилируется, если `id` был просто определен как `val`-переменная с ключевым словом `this`, указанным в правой части, поскольку в этом случае компилятор Scala будет рассматривать `id` в качестве обычного идентификатора.

Подобный синтаксис уже встречался в разделе 29.4, где использовался для предоставления трейту `self`-типа. Псевдонимы также могут послужить неплохой сокращенной записью, когда понадобится обратиться с помощью `this` к охватывающему классу. Рассмотрим пример:

```
class Outer { outer =>
```

```
  class Inner {
    println(Outer.this eq outer) // выводит true
  }
}
```

В нем определяются два вложенных класса, `Outer` и `Inner`. Внутри `Inner` на значение `this` класса `Outer` имеются две ссылки с использованием разных выражений. В первом выражении показан способ, применяемый в Java: перед зарезервированным словом `this` можно поставить имя внешнего класса и точку, и такое выражение затем сошлется на `this` внешнего класса. Во втором выражении показана альтернатива, предоставляемая языком Scala. За счет введения псевдонима, названного `outer` для `this` в классе `Outer`, появляется возможность ссылаться на псевдоним `this` непосредственно во внутренних классах. Способ, предлагаемый в Scala, гораздо лаконичнее и может повысить разборчивость кода при удачном выборе псевдонима. Примеры применения таких псевдонимов будут показаны чуть позже.

Парсеры для анализа отдельно взятых токенов

В трейте `Parsers` определен универсальный парсер `elem`, который может использоваться для анализа любого отдельно взятого токена:

```
def elem(kind: String, p: Elem => Boolean) =
  new Parser[Elem] {
    def apply(in: Input) =
      if (p(in.first)) Success(in.first, in.rest)
      else Failure(kind + " expected", in)
  }
```

Этот парсер получает два параметра: строку `kind`, являющуюся описанием разновидности анализируемого токена, и

применяемый к Elem-объектам предикат p, который указывает, соответствует ли элемент классу токенов, подлежащих парсингу.

Когда парсер elem(kind, p) применяется к какому-либо вводу in, первый элемент потока входных данных тестируется предикатом p. Если p возвращает true, работа парсера завершается успешно. Его результатом являются сам элемент и оставшийся поток входных данных, начинающийся сразу же после обработанного парсером элемента. Если же p возвращает false, парсер дает сбой с сообщением об ошибке, указывающей, какого рода токен ожидался.

Последовательная композиция

Парсер elem использует только отдельно взятый элемент. Чтобы подвергнуть парсингу более содержательные фразы, можно связать парсеры вместе с помощью оператора последовательной композиции ~. Ранее уже было показано, что P~Q является парсером, который сначала применяет к заданной строке ввода парсер P. Затем, если работа P завершится успешно, к входным данным, оставшимся после того, как свою работу выполнил парсер P, применяется парсер Q.

Комбинатор ~ реализован в виде метода в классе Parser. Его определение показано в листинге 33.6. Метод является элементом класса Parser. Внутри этого класса p указан частью p => в качестве псевдонима this, следовательно, p обозначает левый операнд (или получатель) оператора ~. Его правый операнд представлен-параметром q. Теперь, если последовательность парсеров p~q будет запущена в отношении каких-либо входных данных in, то сначала для in будет запущен p и результатом станет анализ в поиске по шаблону. Если работа p завершится успешно, в отношении оставшихся входных данных in1 будет запущен парсер q. Если работа q также завершится успешно, то успех будет сопутствовать всему парсеру. Его результатом станет ~-объект,

содержащий как результат парсера p (то есть x), так и результат парсера q (то есть y). Если же или p, или q даст сбой, результатом p~q станет объект Failure, возвращенный либо p, либо q.

Листинг 33.6. Метод-комбинатор ~

```
abstract class Parser[+T] ... { p =>
  ...
  def ~ [U](q: => Parser[U]) = new Parser[T~U] {
    def apply(in: Input) = p(in) match {
      case Success(x, in1) =>
        q(in1) match {
          case Success(y, in2) => Success(new ~(x,
y), in2)
         case failure => failure
        }
    case failure => failure
  }
}
```

Результат типа ~ является парсером, возвращающим экземпляр case-класса ~ с элементами типов T и U. Выражение типа T~U является просто более разборчивой сокращенной формой записи для параметризированного типа ~[T, U]. Обычно Scala интерпретирует операции бинарного типа, такие как A op B, как параметризированный тип op[A, B]. Это аналогично ситуации для шаблонов, где бинарный шаблон P op Q также интерпретируется как применение, то есть op(P, Q).

Остальные два оператора последовательных композиций, <~ и ~>, могут быть определены точно так же, как и ~, только с небольшим уточнением порядка вычисления результата. Но изящнее будет определить их в понятиях ~ следующим образом:

```
def <~ [U](q: => Parser[U]): Parser[T] =
  (p~q) ^^ { case x~y => x }
def ~> [U](q: => Parser[U]): Parser[U] =
  (p~q) ^^ { case x~y => y }
```

Альтернативная композиция

В альтернативной композиции `P | Q` к заданным входным данным применяется либо `P`, либо `Q`. Сначала предпринимается попытка использования `P`. Если работа `P` завершается успешно, то завершается и работа всего парсера с выдачей результата `P`. Но если `P` даст сбой, предпринимается попытка использования в отношении тех же входных данных парсера `Q`. Затем результат выполнения `Q` становится результатом выполнения всего парсера.

Определение метода `|` класса `Parser` выглядит следующим образом:

```
def | (q: => Parser[T]) = new Parser[T] {
  def apply(in: Input) = p(in) match {
    case s1 @ Success(_, _) => s1
    case failure => q(in)
  }
}
```

При сбое обоих парсеров, и `P`, и `Q`, сообщение об ошибке определяется парсером `Q`. Этот неочевидный выбор будет рассмотрен в разделе 33.9.

Работа с рекурсией

Обратите внимание на то, что параметр `q` в методах `~` и `|` является параметром до востребования — перед его типом стоит стрелка `=>`. Это означает, что фактический аргумент парсера будет вычислен лишь в том случае, если понадобится `q`, а это случится только после

запуска p. Благодаря этому появляется возможность создания рекурсивных парсеров, похожих на следующий, который выполняет парсинг числа, заключенного в произвольное количество скобок:

```
def parens = floatingPointNumber | "("~parens~")"
```

Если | и ~ получили *параметры до востребования*, это определение тут же вызовет переполнение стека, еще ничего не считывая, поскольку значение parens оказалось в середине его правой стороны.

Преобразование результата

Последний метод класса Parser преобразует результат парсера. Парсер P €€ f завершает свою работу успешно при успешном завершении работы парсером P. В таком случае данный парсер возвращает результат парсера P, преобразованный с помощью функции f. Реализация этого метода выглядит следующим образом:

```
  def ^^ [U](f: T => U): Parser[U] = new Parser[U]
{
    def apply(in: Input) = p(in) match {
      case Success(x, in1) => Success(f(x), in1)
      case failure => failure
    }
  }
} // завершение Parser
```

Парсеры, не считывающие никаких входных данных

Существуют еще два парсера, которые не используют никаких входных данных: success и failure. Парсер success(result)

всегда завершает работу успешно с заданным результатом `result`. Парсер `failure(msg)` всегда дает сбой с выдачей сообщения об ошибке `msg`. Оба они реализованы в виде методов в трейте `Parsers`, во внешнем трейте, который также содержит класс `Parser`:

```
def success[T](v: T) = new Parser[T] {
  def apply(in: Input) = Success(v, in)
}
def failure(msg: String) = new Parser[Nothing] {
  def apply(in: Input) = Failure(msg, in)
}
```

Возвращение Option-значения и повторение

В трейте `Parsers` также реализованы комбинаторы возвращения `Option`-значения и повторения `opt`, `rep` и `repsep`. Они реализованы через последовательные композиции, альтернативные варианты и преобразования результата:

```
  def opt[T](p: => Parser[T]): Parser[Option[T]] =
(
    p ^^ Some(_)
  | success(None)
  )

  def rep[T](p: => Parser[T]): Parser[List[T]] = (
    p~rep(p) ^^ { case x~xs => x :: xs }
  | success(List())
  )

  def repsep[T](p: => Parser[T],
      q: => Parser[Any]): Parser[List[T]] = (
```

```
      p~rep(q ~> p) ^^ { case r~rs => r :: rs }
    | success(List())
    )
  } // завершение Parsers
```

33.7. Строковые литералы и регулярные выражения

Показанные до сих пор парсеры использовали для парсинга отдельных слов строковые литералы и регулярные выражения. Поддержка этих возможностей исходит из `RegexParsers`, подтрейта `Parsers`:

```
trait RegexParsers extends Parsers {
```

Этот трейт имеет более специализированный характер, чем трейт `Parsers`, в том смысле, что он работает только с входными данными, представляющими собой последовательность символов:

```
type Elem = Char
```

В нем определяются два метода, `literal` и `regex`, имеющие следующие сигнатуры:

```
implicit def literal(s: String): Parser[String] =
...
implicit def regex(r: Regex): Parser[String] = ...
```

Заметьте, что у обоих методов имеется модификатор `implicit`, следовательно, они автоматически применяются там, где задаются значения типа `String` или `Regex`, но ожидается значение `Parser`. Именно поэтому допускается непосредственное написание в грамматике строковых литералов и регулярных выражений без необходимости заключения их в один из этих методов. Например, парсер `"("~expr~")"` будет автоматически развернут в `literal("(")~expr~literal(")")`.

В трейте RegexParsers также предусмотрена обработка между обозначениями пробельных символов. Для этого перед запуском парсера literal или regex в трейте вызывается метод по имени handleWhiteSpace. Он пропускает самую длинную последовательность входных данных, соответствующую регулярному выражению whiteSpace, которое в исходном виде определяется следующим образом:

```
protected val whiteSpace = """\s+""".r  } // завершение RegexParsers
```

Если предпочтение отдается иной трактовке пробельных символов, значение val-переменной whiteSpace можно переписать. Например, если нужно, чтобы пробельные символы вообще не пропускались, whiteSpace можно переписать в виде пустого регулярного выражения:

```
object MyParsers extends RegexParsers {
  override val whiteSpace = "".r   ...
}
```

33.8. Лексинг и парсинг

Решение задачи выполнения синтаксического анализа зачастую разбивается на две фазы. В фазе лексера во входных данных распознаются отдельные слова и определяется их принадлежность к некоторым классам токенов. Эта фаза называется также лексическим анализом. За ней следует фаза синтаксического анализа, в которой анализируется последовательность токенов. Синтаксический анализ иногда называют парсингом, хотя это не совсем верно, поскольку лексический анализ также может быть отнесен к задаче парсинга.

В соответствии с описанием, рассмотренным в предыдущем разделе, трейт Parsers может использоваться на любой из фаз,

поскольку его входные элементы принадлежат к абстрактному типу `Elem`. Для лексического анализа из данных типа `Elem` могут создаваться экземпляры типа `Char`, означающие отдельные символы, составляющие слово, подвергаемое парсингу. В свою очередь для синтаксического анализа из данных типа `Elem` могут создаваться экземпляры типа токенов, возвращаемых лексером.

Парсер-комбинаторы Scala предоставляют для лексического и синтаксического анализа несколько полезных классов. Они содержатся в двух подпакетах отдельно для каждой разновидности анализа:

```
scala.util.parsing.combinator.lexical
scala.util.parsing.combinator.syntactical
```

Если нужно разбить парсер отдельно на лексер и синтаксический анализатор, можно обратиться к документации по этим пакетам в Scaladoc. Но что касается простых парсеров, обычно бывает достаточно воспользоваться ранее показанным в этой главе подходом на основе регулярного выражения.

33.9. Сообщения об ошибках

Есть еще одна последняя и пока не рассмотренная тема: как парсер выдает сообщение об ошибке? Выдача сообщений об ошибках для парсеров сродни черной магии. Одна из проблем заключается в том, что парсер, отвергая входные данные, обычно обнаруживает множество различных недочетов. Неудачный исход должен иметься у каждого альтернативного синтаксического анализа, и рекурсивно это случается в любой точке выбора. Тогда какой именно из многочисленных сбоев должен выдаваться пользователю в сообщении об ошибке?

В имеющейся в Scala библиотеке парсинга реализовано простое эвристическое правило: среди всех сбоев выбирается тот, который произошел на последней позиции входных данных. Иными

словами, парсер выбирает самый длинный приемлемый префикс и выдает сообщение об ошибке с описанием того, почему синтаксический анализ префикса не может быть продолжен. Если в последней позиции имеется сразу несколько точек сбоя, выбирается та из них, к которой было последнее обращение.

Рассмотрим, к примеру, работу JSON-парсера над дефектной адресной книгой, которая начинается со следующей строки:

```
{ "name": John,
```

Самый длинный приемлемый префикс в этой фразе имеет вид `{ "name":`. Следовательно, JSON-парсер поставит флажок, отмечая ошибку, на имени `John`. В данном месте JSON-парсер ожидал значения, но `John` является идентификатором, не содержащим значения (по-видимому, автор документа забыл заключить имя в кавычки). Сообщение об ошибке, выданное парсером для этого документа, будет иметь следующий вид:

```
[1.13] failure: "false" expected but identifier
John found
  { "name": John,
            ^
```

Предполагаемая сбойная часть берется на основе того факта, что `"false"` в грамматике JSON является последним альтернативным вариантом правила для `value`. Следовательно, данный сбой и выступил в качестве последнего в этом месте. Пользователи, разбирающиеся в тонкостях грамматики JSON, могут реконструировать сообщение об ошибке, но у тех, кто в них не разбирается, данное сообщение может, наверное, вызвать удивление или ввести в заблуждение.

Более полезное сообщение об ошибке может быть создано добавлением в последнюю альтернативу правила `value`, обобщающего места сбоя:

```
def value: Parser[Any] =
   obj | arr | stringLit | floatingPointNumber |
"null" |
   "true" | "false" | failure("illegal start of
value")
```

Это добавление не изменяет набор входных данных, приемлемых в качестве допустимого документа. Оно просто повышает полезность сообщений об ошибках, поскольку теперь явным образом добавляется сбой, получающийся в последнем альтернативном варианте и поэтому отображаемый в отчете об ошибке:

```
[1.13] failure: illegal start of value
  { "name": John,
            ^
```

Для того чтобы пометить сбой, произошедший в позиции входных данных, в реализации последней возможной схемы отчета об ошибках используется поле `lastFailure` в трейте `Parsers`:

```
var lastFailure: Option[Failure] = None
```

Поле инициализируется значением `None`. Оно обновляется в конструкторе класса `Failure`:

```
case class Failure(msg: String, in: Input)
    extends ParseResult[Nothing] {
  if (lastFailure.isDefined &&
      lastFailure.get.in.pos <= in.pos)
    lastFailure = Some(this)
}
```

Значение поля считывается методом `phrase`, который при сбое парсера выдает конечное сообщение об ошибке. Реализация

метода phrase в трейте Parsers выглядит следующим образом:

```
def phrase[T](p: Parser[T]) = new Parser[T] {
  lastFailure = None
  def apply(in: Input) = p(in) match {
    case s @ Success(out, in1) =>
      if (in1.atEnd) s
      else Failure("end of input expected", in1)
    case f : Failure =>
      lastFailure
  }
}
```

Метод phrase запускает свой аргумент — парсер p. Если работа p завершается успешно с полностью использованными входными данными, возвращается успешный результат работы парсера p. Если работа p завершается успешно, но входные данные считаны не полностью, выдается сообщение о сбое с формулировкой end of input expected («Ожидалось окончание входных данных»). Если работа p завершается сбоем, возвращаются сведения о сбое или ошибке, сохраненные в поле lastFailure. Следует заметить, что подход, примененный при реализации поля lastFailure, нефункционален — его значение обновляется конструктором класса Failure и самим методом phrase благодаря побочному эффекту. Можно было бы создать и функциональную версию той же самой схемы, но для нее потребовалось бы распараллелить значение lastFailure на каждый результат парсера независимо от того, к какому бы типу он относился, Success или Failure.

33.10. Сравнение отката с LL(1)

Для выбора между альтернативными парсерами парсер-комбинаторы применяют *откат*. Если фигурирующий в

выражении `P | Q` парсер `P` дает сбой, запускается парсер `Q`, использующий те же самые входные данные, что и `P`. Это происходит, даже если до сбоя `P` уже выполнил парсинг некоторых токенов. В таком случае те же самые токены пройдут парсинг еще раз, но уже с использованием парсера `Q`.

С целью сохранения возможности парсинга откат накладывает на способ формулировки грамматики некоторые ограничения. По сути, нужно всего лишь избегать леворекурсивных правил. Такой набор правил, как

```
expr ::= expr "+" term | term.
```

всегда будет приводить к сбою, поскольку выражение expr тут же вызывает само себя и поэтому никогда не продвигается дальше[152]. В то же время откат потенциально затратен, так как одни и те же входные данные могут быть проанализированы несколько раз. Рассмотрим, к примеру, следующий набор правил:

```
expr ::= term "+" expr | term.
```

Что произойдет, если парсер expr применяется к таким входным данным, как `(1 + 2) * 3`, представляющим собой вполне легальный терм? Сначала будет применен первый вариант альтернативы, который даст сбой при поиске соответствия знаку `+`. Затем в отношении того же самого терма будет применен второй альтернативный вариант, работа которого завершится успешно. В итоге разбор терма окажется выполненным дважды.

Зачастую грамматику можно изменить таким образом, чтобы получить возможность избавиться от отката. Например, в случае арифметических выражений сработает одно из двух правил:

```
expr ::= term [ "+" expr].
expr ::= term { "+" term}.
```

Многие языки допускают использование так называемых LL(1)-

грамматик[153]. Когда парсер-комбинатор сформирован на основе такой грамматики, откаты исключены, то есть позиция входа никогда не будет заново установлена на более раннее значение. Например, к LL(1) относятся показанные ранее в этой главе грамматики и для арифметических выражений, и для термов JSON, поэтому возможность отката для входных данных из этих языков в среде парсер-комбинаторов никогда не используется.

Работа среды парсер-комбинаторов позволяет высказать предположение о принадлежности грамматики к LL(1) явным образом, используя новый оператор `~!`. Этот оператор похож на оператор последовательной композиции `~`, но он никогда не приводит к откату к непрочитанным элементам входных данных, которые уже подвергались парсингу. При использовании этого оператора правила в парсере арифметических выражений могут быть переписаны следующим образом:

```
def expr : Parser[Any] =
  term ~! rep("+" ~! term | "-" ~! term)
def term : Parser[Any] =
  factor ~! rep(" * " ~! factor | "/" ~! factor)
def factor: Parser[Any] =
  "(" ~! expr ~! ")" | floatingPointNumber
```

Одно из преимуществ LL(1)-парсера заключается в том, что он может использовать более простую технологию обработки входных данных. Эти данные могут быть считаны последовательно, и входные элементы после считывания можно отбросить. В этом заключается еще одна причина, по которой LL(1)-парсеры обычно более эффективны по сравнению с парсерами, использующими откаты.

Резюме

Теперь вы узнали обо всех наиболее важных элементах имеющейся в Scala среды парсер-комбинаторов. Для такого по-настоящему полезного средства в них содержится на удивление мало кода. Используя среду, можно создавать парсеры для большого класса бесконтекстных грамматик. Среда позволяет незамедлительно начинать ее использовать, сохраняя при этом возможность настройки под новые разновидности грамматик и методы ввода данных. Будучи библиотекой Scala, эта среда безо всяких проблем интегрируется во все остальные составляющие языка. Таким образом, парсер-комбинаторы довольно легко интегрируются в более объемные программы.

Один из недостатков парсер-комбинаторов состоит в их невысокой эффективности, по крайней мере по сравнению с парсерами, создаваемыми с применением таких средств специального назначения, как Yacc или Bison. Причины здесь две. Во-первых, это невысокая эффективность самого метода отката, используемого парсер-комбинатором. Из-за повторяющихся откатов в зависимости от структуры грамматики и входных данных парсера скорость работы может экспоненциально снижаться. Проблема может быть устранена приведением грамматики к виду LL(1) и тем, что предпочтение будет отдано использованию оператора последовательной композиции ~!.

Второй проблемой, отрицательно влияющей на производительность парсер-комбинаторов, является присущее им смешивание конструкции парсера и анализа входных данных в одном наборе операций. По сути, для каждого анализируемого входа парсер создается заново.

С этой проблемой можно справиться, но для этого потребуется иная реализация среды парсер-комбинаторов. В оптимизированной среде для получения результатов синтаксического анализа парсер больше не будет представлен функцией от входных данных. Вместо этого он будет представлен деревом, где каждый шаг истолкования будет представлен как case-класс. Например, последовательная композиция может быть

представлена case-классом `Seq`, альтернатива — классом `Alt` и т. д. Внешний метод парсинга, `phrase`, тогда сможет получить символьное представление парсера и превратить его в высокоэффективные таблицы парсинга с использованием стандартных алгоритмов парсер-генераторов.

Самое приятное здесь то, что с точки зрения пользователя по сравнению с использованием обычных парсер-комбинаторов ничего не изменится. Пользователи по-прежнему будут создавать парсеры в понятиях `ident`, `floatingPointNumber`, `~`, `|` и т. д. Им не потребуется вникать в то, что эти методы создают символьное представление парсера, а не функцию парсера. Поскольку `phrase`-комбинатор превращает эти представления в реальные парсеры, все будет работать, как и прежде.

Что касается производительности, при такой схеме будет получено двойное преимущество. Во-первых, теперь любую конструкцию парсера можно будет вынести за скобки анализа входных данных. Если бы нужно было воспользоваться таким кодом:

```
val jsonParser = phrase(value)
```

а затем применить `jsonParser` к нескольким различным входным данным, `jsonParser` создавался бы только один раз, а не при каждом считывании входных данных.

Во-вторых, в работе парсер-генератора можно будет воспользоваться эффективными алгоритмами парсинга, такими как LALR(1)[154]. Эти алгоритмы обычно позволяют существенно ускорить процесс парсинга по сравнению с парсингом с откатом.

На данный момент такой оптимизированный парсер-генератор для Scala еще не создан. Но возможность его разработки все же существует. Если кто-нибудь сделает такой генератор, то его можно будет легко интегрировать в стандартную библиотеку Scala. Но если даже предположить, что в будущем такой генератор появится, повод для сохранения текущей среды парсер-

комбинаторов все же останется. Он намного проще для понимания и настройки, чем парсер-генератор, а разница в скорости работы на практике зачастую не играет существенной роли, если только вы не собираетесь выполнять синтаксический анализ очень большого объема входных данных.

152 Существуют способы, позволяющие избегать переполнения стека даже при наличии леворекурсивного правила, но они требуют более качественной среды парсинг-комбинаторов, которая на данный момент еще не реализована.

153 Aho A.V., Sethi R., Ullman J. Compilers: Principles, Techniques, and Tools. — Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 1986.

154 *Aho, et al.* Compilers: Principles, Techniques, and Tools.

34. Программирование GUI

В этой главе мы рассмотрим способы разработки Scala-приложений, использующих графический пользовательский интерфейс (graphical user interface (GUI)). Намеченные к разработке приложения основаны на библиотеке Scala, предоставляющей доступ к имеющейся в Java среде Swing-классов GUI. Концептуально библиотека Scala похожа на положенные в ее основу Swing-классы, но при этом она скрывает большинство их сложных моментов. Благодаря этому вы поймете, что разработка GUI-приложений с использованием данной среды дается без особых затруднений.

Даже при характерных для Scala упрощениях среда, подобная Swing, перенасыщена всевозможными классами, в каждом из которых определено множество методов. Найти свой путь в столь богатой библиотеке поможет использование такой IDE-среды, как имеющееся в Scala расширение Eclipse. Очевидное его преимущество заключается в том, что среда IDE может в интерактивном режиме выполнения своих команд показать, какие классы доступны в пакете и какие методы доступны объектам, на которые вы ссылаетесь. Это существенно ускорит освоение ранее неизвестного библиотечного пространства.

34.1. Первое Swing-приложение

Первым Swing-приложением, с которого мы начнем работу, станет окно, содержащее одну кнопку. Для программирования с использованием Swing нужно будет импортировать из имеющегося в Scala пакета Swing API различные классы:

```
import scala.swing._
```

Код вашего первого Swing-приложения на Scala показан в

листинге 34.1.

Листинг 34.1. Простое Swing-приложение на Scala

```
import scala.swing._

object FirstSwingApp extends
SimpleSwingApplication {
  def top = new MainFrame {
    title = "First Swing App"
    contents = new Button {
      text = "Click me"
    }
  }
}
```

Если откомпилировать этот файл и запустить его на выполнение, вы увидите окно, показанное в левой части рис. 34.1. Как видно в правой части этого рисунка, размеры окна можно увеличить.

Рис. 34.1. Простое Swing-приложение: исходный вид *(слева)* и увеличенный вид *(справа)*

Если построчно проанализировать код листинга 34.1, можно заметить следующие элементы:

```
object FirstSwingApp extends
SimpleSwingApplication {
```

В первой строке после инструкции `import` объект `FirstSwingApp` является наследником класса `scala.swing.SimpleSwingApplication`. Это приложение отличается от традиционного приложения командной строки, которое может быть наследником класса `scala.App`. Класс `SimpleSwingApplication` уже определяет метод `main`, содержащий настроечный код для Java-среды Swing. Затем метод `main` продолжается вызовом предоставляемого вами метода `top`:

```
def top = new MainFrame {
```

В следующей строке реализуется метод `top`. Он содержит код, определяющий ваш GUI-компонент верхнего уровня. Обычно это некая разновидность фрейма `Frame`, то есть окна, которое может содержать произвольные данные. В листинге 34.1 в качестве компонента верхнего уровня был выбран `MainFrame`, который похож на обычный Swing `Frame`, за исключением того, что с его закрытием закрывается и все GUI-приложение:

```
title = "First Swing App"
```

У `Frame`-объектов имеется ряд атрибутов. Двумя наиболее важными атрибутами являются заголовок фрейма `title`, который будет записан в заголовке открывающегося окна, и его содержимое `content`, которое будет показано в самом окне. В имеющемся в Scala Swing API такие атрибуты смоделированы в виде свойств. Из материалов раздела 18.2 известно, что свойства кодируются в Scala в виде пар методов получения (get-методов) и присваивания значения (set-методов). Например, свойство `title` объекта типа `Frame` моделируется в виде метода получения значения:

```
def title: String
```

и метода присваивания значения:

```
def title_=(s: String)
```

Это тот самый метод присваивания, который вызывается показанным ранее присваиванием значения переменной `title`. Эффект присваивания состоит в том, что выбранный заголовок отображается в заголовке окна. Если этот код убрать, у окна будет пустой заголовок.

```
contents = new Button {
```

Корневым компонентом Swing-приложения является фрейм `top`. Это контейнер (`Container`), следовательно, все остальные компоненты должны быть определены именно в нем. Каждый Swing-контейнер обладает свойством `contents`, позволяющим получать и устанавливать содержащиеся в нем компоненты. Метод получения значения свойства `contents` относится к типу `Seq[Component]`, показывающему, что содержимым компонента в целом могут быть сразу несколько объектов. Но в качестве содержимого `Frame`-объектов всегда фигурирует только один компонент. Значение этому компоненту присваивается с помощью set-метода `contents_=`, и он же потенциально может изменять это значение. Например, в листинге 34.1 содержимое фрейма `top` составляет единственная кнопка `Button`:

```
text = "Click me"
```

У кнопки имеется надпись, в данном случае это `"Click me"`.

34.2. Панели и разметки

Давайте в качестве следующего шага добавим к фрейму `top` нашего приложения текст, который станет вторым элементом его содержимого. Внешний вид, который при этом приобретает приложение, показан в левой части рис. 34.2.

Рис. 34.2. Реагирующее Swing-приложение в исходном состоянии *(слева)* и после щелчков *(справа)*

В предыдущем разделе было показано, что фрейм содержит всего один дочерний компонент. Следовательно, для создания фрейма не только с кнопкой, но и с надписью нужно создать другой контейнерный компонент, содержащий и то и другое. Для этой цели используется панель. `Panel`-объект является контейнером, показывающим все содержащиеся в нем компоненты в соответствии с некими жестко заданными правилами разметки. Существует множество возможных разметок, реализуемых разными подклассами класса `Panel`, от самых простых до весьма замысловатых. По сути, одной из самых сложных частей комплексного GUI-приложения может стать получение правильной разметки: не так уж просто добиться, чтобы что-то довольно рационально отображалось на всех разновидностях устройств и при всех размерах окон.

Полноценная реализация компоновки компонентов показана в листинге 34.2. Имеющиеся в этом классе два подкомпонента фрейма `top` называются `button` и `label`. Подкомпонент `button` определен так же, как в ранее показанном коде. Подкомпонент `label` показывает текстовое поле, не предназначенное для редактирования:

```
val label = new Label {
  text = "No button clicks registered"
}
```

Листинг 34.2. Компоновка компонентов на панели

```
import scala.swing._

object SecondSwingApp extends
SimpleSwingApplication {
  def top = new MainFrame {
    title = "Second Swing App"
    val button = new Button {
      text = "Click me"
    }
    val label = new Label {
      text = "No button clicks registered"
    }
     contents = new BoxPanel(Orientation.Vertical)
{
      contents += button
      contents += label
      border = Swing.EmptyBorder(30, 30, 10, 30)
    }
  }
}
```

Для кода в листинге 34.2 выбрана простая вертикальная разметка, где компоненты устанавливаются друг на друга в `BoxPanel`:

```
contents = new BoxPanel(Orientation.Vertical) {
```

Свойство `contents`, принадлежащее `BoxPanel`, представляет собой изначально пустой буфер, в который посредством оператора `+=` добавляются элементы `button` и `label`:

```
contents += button
```

```
contents += label
```

С помощью присваивания значения свойству панели `border` была добавлена рамка, окружающая эти два объекта:

```
border = Swing.EmptyBorder(30, 30, 10, 30)
```

Как и другие GUI-компоненты, рамки представлены в виде объектов. `EmptyBorder` является в `Swing`-объекте фабричным методом, получающим четыре параметра, которые показывают ширину рамок рисуемых объектов сверху, справа снизу и слева.

Каким бы простым ни был этот пример, он уже продемонстрировал основной способ структурирования GUI-приложения. Он построен из компонентов, являющихся экземплярами таких классов `scala.swing`, как `Frame`, `Panel`, `Label` или `Button`. У компонентов есть свойства, которые могут настраиваться приложением. Компоненты объекта `Panel` могут содержать в своих свойствах `contents` несколько других компонентов. Таким образом, созданное GUI-приложение состоит из дерева компонентов.

34.3. Обработка событий

От созданного нами приложения, по сути, нет никакого проку. Если запустить на выполнение код, приведенный в листинге 34.2, и нажать показанную в окне кнопку, ничего не произойдет. Фактически приложение имеет статический характер — оно никак не реагирует на события, инициируемые пользователем, за исключением щелчка на кнопке закрытия окна во фрейме `top`, который приводит к завершению работы приложения. Поэтому следующим шагом мы усовершенствуем приложение, чтобы видимая рядом с кнопкой надпись служила индикатором количества нажатий кнопки. В правой части рис. 34.2 содержится копия экрана, показывающая, как должно выглядеть приложение

после нескольких нажатий кнопки.

Чтобы добиться от приложения соответствующего поведения, нужно подключить событие пользовательского воздействия (щелчок на кнопке) к действию (обновлению показываемой надписи). К обработке событий в Java и Scala имеется одинаковый основной подход публикации-подписки: компоненты могут выступать в роли издателей и/или подписчиков. Издатель публикует события. Подписчик подписывается на публикацию, чтобы быть в курсе любых публикуемых событий. Издателей также называют источниками событий, а подписчиков — отслеживателями событий. Например, `Button`-объект является источником события, занимающимся публикацией события нажатия кнопки — `ButtonClicked`.

В Scala подписка на источник события выполняется с помощью вызова метода `listenTo(`источник`)`. Есть также способ отписки от источника события использованием вызова метода `deafTo(`источник`)`. В текущем примере приложения первым делом нужно заставить фрейм `top` отслеживать события вокруг его кнопки, чтобы получать уведомления о любых выдаваемых ею событиях. Для этого нужно добавить к телу фрейма `top` следующий вызов:

```
listenTo(button)
```

Получение уведомления о событии — лишь половина дела, вторая половина заключается в его обработке. Вот здесь среда Scala Swing наиболее существенно отличается от Java Swing API и значительно проще справляется с задачей. В Java появление сигнала о событии означает вызов уведомляющего метода в отношении объекта, который должен реализовать какие-либо `Listener`-интерфейсы. Обычно за всем этим тянется большой объем косвенного и шаблонного кода, затрудняющего написание и чтение приложений, обрабатывающих события. В отличие от этого в Scala событие является настоящим объектом, отправляемым в

адрес подписавшихся компонентов, что во многом похоже на сообщения, отправляемые акторам. Например, нажатие кнопки приведет к созданию события, являющегося экземпляром следующего case-класса:

```
case class ButtonClicked(source: Button)
```

Параметр case-класса ссылается на кнопку, которая была нажата. Как и в случае со всеми другими событиями Scala Swing, этот класс события содержится в пакете `scala.swing.event`.

Чтобы заставить компоненты реагировать на поступающие события, нужно к свойству по имени `reactions` добавить обработчик. Код, выполняющий эту задачу, выглядит следующим образом:

```
var nClicks = 0
reactions += {
  case ButtonClicked(b) =>
    nClicks += 1
      label.text = "Number of button clicks: " +
nClicks
}
```

В первой строке кода определяется переменная `nClicks`, в которой хранятся данные о количестве нажатий кнопки. В остальных строках между фигурными скобками добавляется код, выступающий в качестве *обработчика* свойства `reactions` фрейма `top`. Обработчики являются функциями, определяемыми поиском по шаблону событий, что во многом похоже на то, как актор Akka получает методы, определенные поиском по шаблону сообщений. Показанный ранее обработчик соответствует событиям, имеющим форму `ButtonClicked(b)`, то есть любому событию, являющемуся экземпляром класса `ButtonClicked`. Переменная шаблона `b` ссылается на кнопку, которая была нажата. Действие,

соответствующее этому событию в показанном коде, увеличивает значение nClicks на единицу и обновляет текст надписи.

В общем, обработчик относится к функции PartialFunction, соответствующей событиям и выполняющей некоторые действия. Также в одном обработчике возможно определить соответствия нескольким разновидностям событий, для чего используются несколько case-вариантов.

В свойстве reactions реализуется коллекция, что во многом похоже на то, что выполняется в свойстве contents. Некоторые компоненты берутся из предопределенных реакций. Например, в классе Frame есть предопределенная реакция, закрывающая окно при нажатии пользователем кнопки закрытия в верхнем правом углу. Если установить свои собственные реакции, добавляя их к свойству reactions с помощью оператора +=, то определяемые вами реакции будут рассматриваться наряду со стандартными реакциями. Концептуально обработчики, установленные в reactions, формируют стек. В текущем примере при получении события фреймом top первым будет испробован обработчик, соответствующий ButtonClicked, поскольку он был последним обработчиком, установленным для фрейма. Если полученное событие относится к типу ButtonClicked, вызывается код, связанный с шаблоном. После выполнения кода система станет искать в стеке обработчиков следующие обработчики, которые также можно применить к событию. Если полученное событие не относится к типу ButtonClicked, оно тут же распространяется на все остальное, что установлено в стеке обработчиков. Также обработчики из свойства reactions можно удалять, воспользовавшись для этого оператором -=.

В листинге 34.3 показано полноценное приложение, включающее реакции. В коде проиллюстрированы основные элементы GUI-приложения в среде Scala Swing: приложение состоит из трех компонентов, начинающихся с фрейма top. В коде показаны компоненты Frame, BoxPanel, Button и Label, но

существует также множество других разновидностей компонентов, определенных в библиотеках Swing.

Листинг 34.3. Реализация реагирующего на события Swing-приложения

```
import scala.swing._
import scala.swing.event._

object ReactiveSwingApp extends
SimpleSwingApplication {
  def top = new MainFrame {
    title = "Reactive Swing App"
    val button = new Button {
      text = "Click me"
    }
    val label = new Label {
      text = "No button clicks registered"
    }
     contents = new BoxPanel(Orientation.Vertical)
{
      contents += button
      contents += label
      border = Swing.EmptyBorder(30, 30, 10, 30)
    }
    listenTo(button)
    var nClicks = 0
    reactions += {
      case ButtonClicked(b) =>
        nClicks += 1
         label.text = "Number of button clicks: " +
nClicks
```

```
      }
    }
  }
```

Каждый компонент настраивается установочными атрибутами. Два важных атрибута — `contents`, в котором фиксируются дочерние элементы компонента в дереве, и `reactions`, в котором определяются способы реагирования компонента на события.

34.4. Пример: программа перевода градусов между шкалами Цельсия и Фаренгейта

В качестве примера создадим GUI-программу перевода градусов между шкалами Цельсия и Фаренгейта. Пользовательский интерфейс программы представлен на рис. 34.3. Он состоит из двух текстовых полей, показанных на белом фоне, за каждым из которых следует надпись. В одном текстовом поле отображается температура в градусах по Цельсию, а в другом — в градусах по Фаренгейту. Пользователь приложения может редактировать каждое из полей. Как только он меняет температуру в любом из полей, автоматически обновляется температура в другом поле.

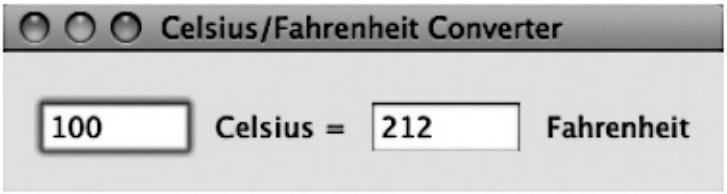


Рис. 34.3. Программа перевода градусов между шкалами Цельсия и Фаренгейта

Полный код, реализующий это приложение, показан в листинге 34.4. Его важной составляющей является использование в верхней части сокращенных записей

```
import swing._
```

```
import event._
```

Фактически они являются эквивалентами импортирования, которое использовалось ранее:

```
import scala.swing._
import scala.swing.event._
```

Листинг 34.4. Реализация температурного конвертора

```
import swing._
import event._

object TempConverter extends
SimpleSwingApplication {
  def top = new MainFrame {
    title = "Celsius/Fahrenheit Converter"
    object celsius extends TextField { columns = 5
}
     object fahrenheit extends TextField { columns
= 5 }
    contents = new FlowPanel {
      contents += celsius
      contents += new Label(" Celsius = ")
      contents += fahrenheit
      contents += new Label(" Fahrenheit")
      border = Swing.EmptyBorder(15, 10, 10, 10)
    }
    listenTo(celsius, fahrenheit)
    reactions += {
      case EditDone('fahrenheit') =>
        val f = fahrenheit.text.toInt
        val c = (f - 32) * 5 / 9
```

```
          celsius.text = c.toString
        case EditDone('celsius') =>
          val c = celsius.text.toInt
          val f = c * 9 / 5 + 32
          fahrenheit.text = f.toString
      }
    }
  }
```

Возможность использования сокращенной записи обусловливается тем, что эти пакеты вложены в Scala. Поскольку пакет `scala.swing` содержится в пакете `scala` и все содержимое пакета импортируется автоматически, то для ссылки на пакет можно просто записать `swing`. Аналогично этому пакет `scala.swing.event` содержится в качестве подпакета `event` в пакете `scala.swing`. Поскольку благодаря первой инструкции `import` вы уже импортировали все, что имеется в `scala.swing`, то после этого на пакет `event` можно просто сослаться как на `event`.

Два компонента, `celsius` и `fahrenheit`, в `TempConverter` являются объектами класса `TextField`. Класс `TextField` в Swing выступает компонентом, позволяющим редактировать одну строку текста. Его исходная ширина задается свойством `columns`, значение которого измеряется в символах (для обоих полей установлено значение 5).

Содержимое `TempConverter` собрано на панели, куда включены два текстовых поля и две надписи, поясняющие, что это за поля. Панель относится к классу `FlowPanel`, а это означает, что она отображает все свои элементы один за другим в одной или нескольких строках в зависимости от ширины фрейма.

Свойство `reactions`, принадлежащее `TempConverter`, определено обработчиком, содержащим два case-варианта. Каждый case-вариант соответствует событию `EditDone` для одного из двух текстовых полей. Это событие выдается после

редактирования текстового поля пользователем. Обратите внимание на форму шаблонов, включающую обратные кавычки вокруг имен элементов:

```
case EditDone(`celsius`)
```

В соответствии с разъяснениями, которые были даны в разделе 15.2, обратные кавычки вокруг `celsius` гарантируют, что соответствие шаблону будет определяться только в том случае, если источником события станет объект `celsius`. Если обратные кавычки не поставить, а просто написать `case EditDone(celsius)`, шаблон будет соответствовать каждому событию класса `EditDone`. Затем измененное поле будет сохранено в переменной шаблона `celsius`. Очевидно, это совсем не то, что вам было нужно. В качестве альтернативного варианта можно определить два объекта `TextField` с именами, начинающимися с символов в верхнем регистре, то есть `Celsius` и `Fahrenheit`. В таком случае соответствие им можно определять напрямую, без обратных кавычек, как в коде `case EditDone(Celsius)`.

Два действия событий `EditDone` преобразуют одну величину в другую. Каждое из них начинается со считывания содержимого измененного поля и его преобразования в `Int`-значение. Затем для преобразования одного температурного показателя в градусах в другой применяется формула, а результат сохраняется как строка в другом текстовом поле.

Резюме

В этой главе вам была предоставлена возможность получить свои первые впечатления о GUI-программировании с использованием имеющейся в Scala оболочки для среды Swing. В ней было показано, как собираются GUI-компоненты, как можно настроить их свойства и как обрабатываются события. Из-за экономии места

в книге мы смогли рассмотреть лишь небольшое количество простых компонентов. Но существует множество компонентов других разновидностей. Узнать о них можно из документации по языку Scala, касающейся пакета `scala.swing`. В следующей главе будет представлен пример разработки более сложного Swing-приложения.

Имеется также множество руководств по исходной среде Java Swing, на которой основана имеющаяся в Scala оболочка[155]. Оболочки Scala похожи на исходные Swing-классы, но в них предпринята попытка упрощения концепций и придания им более универсального характера. В упрощениях широко используются свойства языка Scala. Например, имеющаяся в Scala эмуляция свойств и перегрузка операторов позволяют составлять удобные определения свойств, применяя присваивания и операции `+=`. Используемая в языке философия «абсолютно все является объектом» позволяет наследовать метод `main` GUI-приложения. Таким образом, этот метод может быть скрыт от пользовательских приложений, включая весь характерный для него рутинный настроечный код. И наконец, самое главное: имеющиеся в Scala функции первого класса и технология поиска по шаблону позволяют формулировать обработку событий в виде свойства компонента `reactions`, тем самым существенно упрощая жизнь разработчику приложений.

155 Обратите, к примеру, внимание на издание The Java Tutorials.

35. Электронная таблица SCells

В предыдущих главах было рассмотрено множество различных конструкций языка программирования Scala. Здесь же будет показана совместная работа этих конструкций при реализации серьезного приложения. Задача заключается в создании приложения в виде электронной таблицы, для которого выбрано имя SCells.

Это приложение представляет интерес по нескольким причинам. Во-первых, все знают, что такое электронные таблицы, следовательно, будет нетрудно разобраться в том, чем именно должно заниматься приложение. Во-вторых, электронные таблицы являются программами, в которых выполняется широкий диапазон различных вычислительных задач. Присутствует визуальный аспект: электронная таблица представляется как весьма развитое GUI-приложение. Есть символьный аспект, касающийся формул, их синтаксического анализа и интерпретации. Имеется также вычислительный аспект, предусматривающий наращиваемое обновление, возможно, весьма крупных таблиц. Не обходится и без аспекта реагирования, выражающегося в подходе к электронным таблицам как к программам, реагирующим на события довольно сложным образом. И наконец, существует аспект компонентов, при котором приложение конструируется в виде набора компонентов, допускающих многократное использование. И все эти аспекты будут рассматриваться в этой главе довольно глубоко.

35.1. Визуальная среда

Начнем с создания основной визуальной среды приложения. В первом приближении внешний вид пользовательского интерфейса показан на рис. 35.1. Как видите, электронная таблица может прокручиваться. В ней имеются строки от 0 до 99 и столбцы от A до Z. В Swing это выражается путем определения электронной

таблицы в виде объекта ScrollPane, содержащего объект Table. Код показан в листинге 35.1.

Листинг 35.1. Код для электронной таблицы, показанной на рис. 35.1

```
package org.stairwaybook.scells
import swing._

class Spreadsheet(val height: Int, val width: Int)
    extends ScrollPane {

  val table = new Table(height, width) {
    rowHeight = 25
    autoResizeMode = Table.AutoResizeMode.Off
    showGrid = true
    gridColor = new java.awt.Color(150, 150, 150)
  }

  val rowHeader =
          new  ListView((0  until  height)  map
(_.toString)) {
      fixedCellWidth = 30
      fixedCellHeight = table.rowHeight
    }
  viewportView = table
  rowHeaderView = rowHeader
}
```

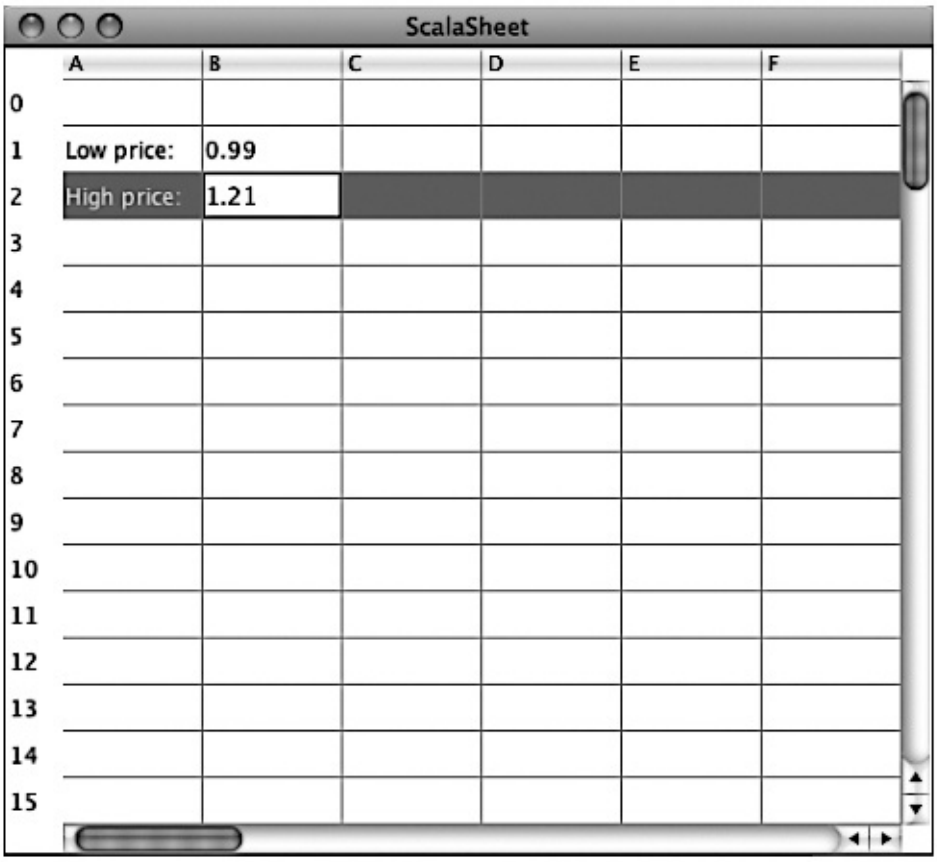


Рис. 35.1. Простая электронная таблица

Компонент электронной таблицы, показанный в листинге 35.1, определен в пакете `org.stairwaybook.scells`, который будет содержать все необходимые приложению классы, трейты и объекты. Основные элементы оболочки Scala Swing импортируются в него из пакета `scala.swing`. Сама электронная таблица является классом, получающим в качестве параметров высоту и ширину (`height` и `width`), которые выражаются в количестве ячеек. Этот класс расширяет класс `ScrollPane`, который дает ему полосы прокрутки, расположенные с правой стороны и внизу таблицы, показанной на рис. 35.1. Он содержит два компонента с именами `table` и `rowHeader`.

Компонент `table` является экземпляром безымянного подкласса, родительским классом которого является `scala.swing.Table`. В четырех строках его тела устанавливаются

значения его атрибутов: rowHeight для высоты строки таблицы в точках, autoResizeMode для отключения автоматической установки размеров таблицы, showGrid для показа сетки между ячейками и gridColor для задания сетке темно-серого цвета.

Компонент rowHeader, содержащий заголовки с номерами строк в левой части электронной таблицы на рис. 35.1, относится к типу ListView, отображающему в своих элементах строковые значения от 0 до 99. Две строки в его теле фиксируют ширину ячейки, задавая ее значение 30 точек, и высоту, определяя ее равной значению атрибута rowHeight, принадлежащего компоненту table.

Вся электронная таблица собирается установкой значений для двух полей в ScrollPane. Для поля viewportView устанавливается значение table, а для поля rowHeaderView — значение списка rowHeader. Разница между этими двумя представлениями состоит в том, что окно просмотра области прокрутки является областью, прокручиваемой с помощью двух полос, а заголовки строк в левой части при перемещении полосы горизонтальной прокрутки остаются зафиксированными. В соответствии с особенностями Swing заголовки столбцов задаются в исходном виде в верхней части таблицы, что избавляет от необходимости определять их явно.

Чтобы получить простую электронную таблицу, показанную на рис. 35.1, нужно всего лишь определить основную программу, создающую компонент Spreadsheet. Такая программа показана в листинге 35.2.

Листинг 35.2. Основная программа для приложения электронной таблицы

```
package org.stairwaybook.scells
import swing._
```

```
object Main extends SimpleSwingApplication {
  def top = new MainFrame {
    title = "ScalaSheet"
    contents = new Spreadsheet(100, 26)
  }
}
```

Программа `Main` является наследником класса `SimpleSwingApplication`, который берет на себя решение всех низкоуровневых настроек, предваряющих запуск Swing-приложения. Остается только определить высокоуровневое окно приложения в методе `top`. В нашем примере `top` относится к объекту класса `MainFrame`, у которого определяются два элемента: заголовок `title`, для которого установлено значение `ScalaSheet`, и содержимое `contents`, со значением в виде экземпляра класса `Spreadsheet`, имеющего 100 строк и 26 столбцов. Вот и все. Если запустить это приложение командой `scala org.stairwaybook.scells.Main`, появится такая же электронная таблица, что и на рис. 35.1.

35.2. Разделение введенных данных и отображения

Если поработать с созданной на данный момент электронной таблицей, сразу станет заметно, что вывод, отображаемый в ячейке, всегда в точности соответствует тому, что в эту ячейку введено. Настоящая электронная таблица так себя не ведет. В ячейку вводится формула, а отображается вычисленное с ее помощью значение. То есть то, что введено в ячейку, отличается от того, что в ней отображается.

В качестве первого шага к созданию настоящего приложения электронной таблицы нужно сконцентрироваться на разделении введенных данных и отображения. Основной механизм

отображения содержится в методе rendererComponent класса Table. Изначально rendererComponent выводит то, что введено. Если требуется внести изменения в этот порядок, нужно переопределить rendererComponent во что-нибудь другое. Новая версия Spreadsheet с методом rendererComponent показана в листинге 35.3.

Листинг 35.3. Электронная таблица с методом rendererComponent

```
package org.stairwaybook.scells
import swing._

class Spreadsheet(val height: Int, val width: Int)
    extends ScrollPane {

  val cellModel = new Model(height, width)
  import cellModel._

  val table = new Table(height, width) {

  // те же настройки, что и прежде...

       override def rendererComponent(isSelected:
Boolean,
        hasFocus: Boolean, row: Int, column: Int):
Component =
        if (hasFocus) new TextField(userData(row,
column))
      else
         new Label(cells(row)(column).toString) {
           xAlignment = Alignment.Right
         }
```

```
      def userData(row: Int, column: Int): String =
  {
        val v = this(row, column)
        if (v == null) "" else v.toString
      }
    }
    // все остальное, как и прежде...
  }
```

Метод `rendererComponent` переопределяет исходный метод в классе `Table`. Он получает четыре параметра. Параметры `isSelected` и `hasFocus` относятся к типу `Boolean` и показывают, была ли выбрана ячейка и имеет ли она фокус, означающий, что события клавиатуры будут отражаться на содержимом ячейки. Остальные два параметра, `row` и `column`, задают координаты ячейки.

В новом методе `rendererComponent` проверяется наличие у ячейки фокуса ввода. Если значение `hasFocus` равно `true`, значит ячейка используется для редактирования. В таком случае нужно, чтобы отображалось редактируемое поле `TextField`, содержащее данные, введенные пользователем на текущий момент. Эти данные возвращаются благодаря использованию вспомогательного метода `userData`, который отображает содержимое данной строки и данного столбца таблицы. Содержимое извлекается путем вызова `this(row, column)`[156]. Метод `userData` также берет на себя вывод вместо `null`-элемента не null, а пустой строки.

Пока все идет хорошо. Но что будет отображено, если у ячейки нет фокуса? В настоящей электронной таблице будет выводиться значение ячейки. Таким образом, в действительности в работе находятся сразу две таблицы. Первая из них по имени `table` содержит то, что ввел пользователь. Вторая, теневая таблица содержит внутреннее представление ячеек, которое и должно быть отображено. В примере электронной таблицы эта таблица

представлена двумерным массивом cells. Если ячейка на пересечении данных строки и столбца не имеет фокуса редактирования, метод rendererComponent отобразит элемент cells(row)(column). Он не может быть редактируемым, следовательно, должен быть отображен в виде значения типа Label, а не в виде редактируемого значения типа TextField.

Осталось определить внутренний массив ячеек. Это можно сделать непосредственно в классе Spreadsheet, но обычно предпочтительнее отделить представление GUI-компонента от его внутренней модели. Поэтому в показанном ранее примере массив cells определен в отдельном классе по имени Model. Модель интегрирована в Spreadsheet путем определения значения cellModel, относящегося к типу Model. Инструкция import, которая следует за этим определением val-переменной, делает элементы cellModel доступными внутри Spreadsheet без указания для них какого-либо префикса. Первая, упрощенная версия класса Model показана в листинге 35.4. В классе определяются внутренний класс по имени Cell и двумерный массив по имени cells, содержащий элементы типа Cell. Каждый элемент инициализирован как совершенно новый Cell-объект.

Листинг 35.4. Первая версия класса Model

```
package org.stairwaybook.scells

class Model(val height: Int, val width: Int) {

  case class Cell(row: Int, column: Int)

  val cells = Array.ofDim[Cell](height, width)

  for (i <- 0 until height; j <- 0 until width)
```

```
        cells(i)(j) = new Cell(i, j)
    }
```

Пока все правильно. Если откомпилировать измененный класс `Spreadsheet` с классом `Model` и запустить приложение `Main`, появится такое же окно, как на рис. 35.2.

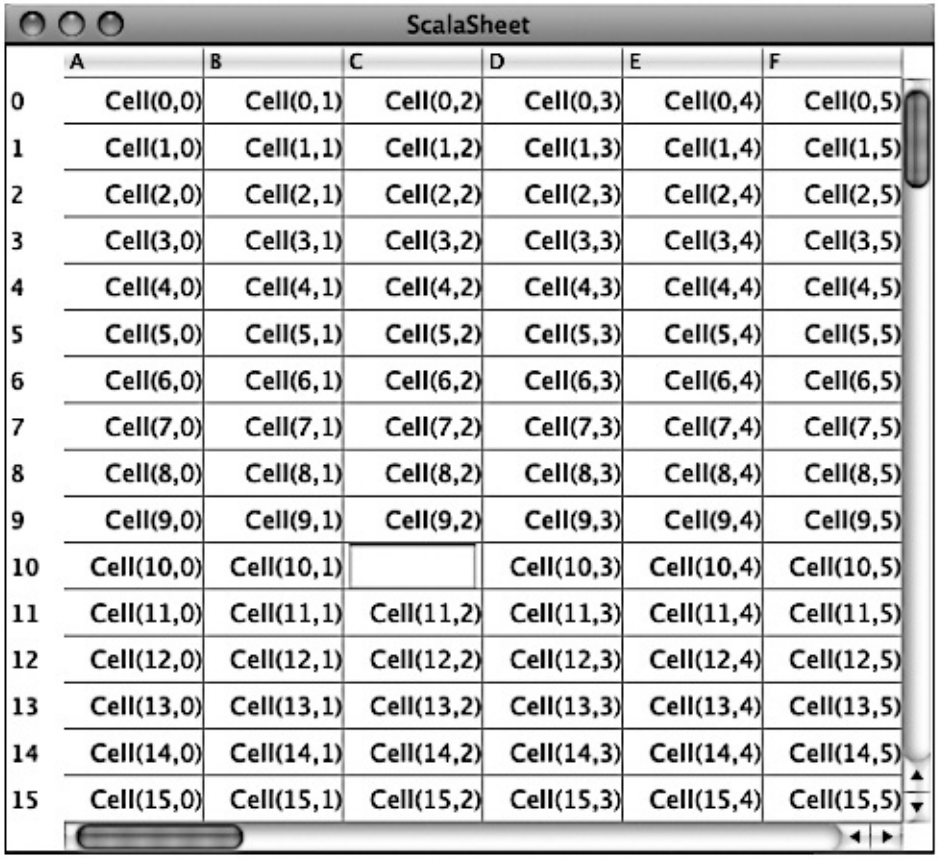


Рис. 35.2. Ячейки, показывающие собственные координаты

Целью данного раздела было создать конструкцию, в которой отображаемое в ячейке значение отличается от введенной в нее строки. Вполне очевидно, что цель эта достигнута, хотя и весьма примитивным способом. В новой электронной таблице можно вводить в ячейку все что угодно, но при потере фокуса в ней всегда будут отображаться только ее координаты. Понятно, что работа еще не завершена.

35.3. Формулы

В действительности в ячейке электронной таблицы хранятся две вещи: текущее значение и формула его вычисления. В электронной таблице имеются три типа формул.

Числовые значения, такие как `1.22`, `-3` или `0`.

46. Текстовые надписи, такие как `Annual sales`, `Deprecation` или `total`.

47. Формулы, вычисляющие новое значение из содержимого ячеек, например `=add(A1,B2)` или `=sum(mul(2, A2), C1:D16)`.

Формула, вычисляющая значение, всегда начинается со знака равенства и продолжается арифметическим выражением. Электронная таблица `SCells` придерживается в отношении арифметических выражений весьма простого и постоянного соглашения: каждое выражение является применением какой-либо функции к списку аргументов. Именем функции выступает идентификатор `add` — для бинарного сложения или `sum` — для суммирования произвольного количества операндов. Аргументом функции может быть число, ссылка на ячейку, ссылка на диапазон ячеек, такая как `C1:D16`, или еще одно применение функции. Чуть позже вы увидите, что у `SCells` имеется открытая архитектура, которая облегчает установку собственных функций посредством составления композиции примесей.

Первым шагом в решении задачи обработки формул станет запись представляющих их типов. В соответствии с возможными ожиданиями различные виды формул представлены case-классами. Содержимое файла **Formulas.scala**, в котором определены эти case-классы, показано в листинге 35.5.

Листинг 35.5. Классы, представляющие формулы

```
package org.stairwaybook.scells

trait Formula

case class Coord(row: Int, column: Int) extends
Formula {
       override def toString = ('A' +
column).toChar.toString + row
}
case class Range(c1: Coord, c2: Coord) extends
Formula {
   override def toString = c1.toString + ":" +
c2.toString
}
case class Number(value: Double) extends Formula {
  override def toString = value.toString
}
case class Textual(value: String) extends Formula
{
  override def toString = value
}
case class Application(function: String,
    arguments: List[Formula]) extends Formula {
  override def toString =
    function + arguments.mkString("(", ",", ")")
}
object Empty extends Textual("")
```

Трейт `Formula`, показанный в листинге 35.5, содержит в качестве дочерних пять case-классов:

- `Coord` — для координат ячейки, например `A3`;
- `Range` — для диапазона ячеек, например `A3:B17`;

- `Number` — для чисел с плавающей точкой, например `3.1415`;
- `Textual` — для текстовых надписей, например `Deprecation`;
- `Application` — для применения функций, например `sum(A1,A2)`.

В каждом case-классе метод `toString` переопределяется с тем, чтобы с его помощью показанным ранее стандартным способом отобразить вид его формул. Для удобства есть также `Empty`-объект, представляющий содержимое пустой ячейки. `Empty`-объект является экземпляром класса `Textual` с пустым строковым аргументом.

35.4. Синтаксический анализ формул

В предыдущем разделе были показаны различные виды формул и порядок их отображения в строковом виде. Здесь же мы поговорим о том, как этот процесс запустить в обратную сторону: преобразовать пользовательский строковый ввод в дерево типа `Formula`. Далее в разделе будут поочередно рассмотрены различные элементы класса `FormulaParsers`, содержащие парсеры, выполняющие преобразования. Класс построен на среде комбинаторов, рассмотренной в главе 33. Выражаясь конкретнее, парсеры формул являются экземплярами класса `RegexParsers`, рассмотренного в этой главе:

```
package org.stairwaybook.scells
import scala.util.parsing.combinator._
object FormulaParsers extends RegexParsers {
```

Первые два элемента объекта `FormulaParsers` являются дополнительными парсерами для идентификаторов и десятичных чисел:

```
def ident: Parser[String] = """[a-zA-Z_]\w * """.r
def decimal: Parser[String] = """-?\d+(\.\d *
)?""".r
```

Если исходить из первого показанного ранее регулярного выражения, то идентификатор начинается с буквы или знака подчеркивания. Затем следует произвольное количество словарных символов, представленных кодом регулярного выражения `\w`, с помощью которого распознаются буквы, цифры или знаки подчеркивания. Второе регулярное выражение дает описание десятичных чисел, в которых могут быть необязательный знак «минус», одна или несколько цифр, представляемых кодом регулярного выражения `\d`, и необязательная дробная часть, состоящая из точки, за которой стоят от нуля до нескольких цифр.

Следующим элементом объекта `FormulaParsers` является парсер ячейки, который распознает координаты ячейки, такие как `C11` или `B2`. Сначала он вызывает парсер на основе регулярного выражения, определяющий форму координат: одна буква, за которой следует одна или несколько цифр. Затем строка, возвращенная этим парсером, преобразуется в координаты ячейки отделением буквы от числовой части и преобразованием двух частей в индексы для столбца и строки ячейки:

```
def cell: Parser[Coord] =
  """[A-Za-z]\d+""".r ^^ { s =>
    val column = s.charAt(0).toUpper - 'A'
    val row = s.substring(1).toInt
    Coord(row, column)
}
```

Обратите внимание на некоторую ограниченность парсера ячейки, позволяющего использовать только те координаты столбцов, которые состоят из одной буквы. Таким образом, количество столбцов электронной таблицы фактически

ограничивается числом не более 26, поскольку последующие столбцы не могут пройти синтаксический анализ. Было бы неплохо расширить возможности парсера, чтобы он воспринимал ячейки с несколькими начальными буквами. Пусть это станет упражнением для самостоятельного выполнения.

Диапазоном, распознаваемым парсером `range`, является диапазон ячеек. Такой диапазон состоит из координат двух ячеек с двоеточием между ними:

```
def range: Parser[Range] =
  cell~":"~cell ^^ {
    case c1~":"~c2 => Range(c1, c2)
  }
```

Парсер числа `number` распознает десятичное число, которое конвертируется в значение типа `Double` и заключается в экземпляр класса `Number`:

```
def number: Parser[Number] =
  decimal ^^ (d => Number(d.toDouble))
```

Парсер применения `application` распознает применение функции. Оно состоит из идентификатора, за которым стоит список аргументов, заключенный в круглые скобки:

```
def application: Parser[Application] =
  ident~"("~repsep(expr, ",")~")" ^^ {
    case f~"("~ps~")" => Application(f, ps)
}
```

Парсер выражения `expr` распознает выражение — либо формулы высшего уровня, следующей за знаком `=`, либо аргумента для функции. Такое выражение формулы определяется в качестве ячейки, диапазона ячеек, числа или применения:

```
def expr: Parser[Formula] =
  range | cell | number | application
```

Это определение парсера `expr` представлено в несколько упрощенном виде, поскольку диапазоны ячеек должны появляться только в аргументах функций — они недопустимы в формулах верхнего уровня. Грамматику формулы можно изменить, чтобы разделить два случая использования выражений, а диапазон синтаксически исключить из формул верхнего уровня. В представленной здесь электронной таблице подобная ошибка обнаруживается при вычислении выражения.

Парсер текста `textual` распознает произвольную строку ввода, если только она не начинается со знака равенства (вспомним, что строки, начинающиеся с `=`, считаются формулами):

```
def textual: Parser[Textual] =
  """[^=]. * """.r ^^ Textual
```

Парсер формул `formula` распознает все виды допустимого для ячейки ввода. Формулой является либо число, либо текстовый ввод, либо формула, начинающаяся со знака равенства:

```
def formula: Parser[Formula] =
  number | textual | "=" ~ >expr
```

На этом грамматика для ячеек электронных таблиц завершается. Заключительный метод `parse` использует эту грамматику в методе, преобразующем строку ввода в дерево типа `Formula`:

```
def parse(input: String): Formula =
  parseAll(formula, input) match {
    case Success(e, _) => e
      case f: NoSuccess => Textual("[" + f.msg +
"]")
```

```
  }
} //окончание FormulaParsers
```

Метод `parse` выполняет синтаксический анализ всего ввода с помощью парсера `formula`. В случае успеха возвращается получившаяся формула. В случае сбоя вместо этого возвращается объект типа `Textual` с сообщением об ошибке.

Вот, собственно, и все, что касается парсинга формул. Осталось только встроить парсеры в электронную таблицу. Для этого можно обогатить класс `Cell` в классе `Model` полем `formula`:

```
case class Cell(row: Int, column: Int) {
  var formula: Formula = Empty
  override def toString = formula.toString
}
```

В новой версии класса `Cell` метод `toString` определен для отображения формулы ячейки. Таким образом, можно будет проконтролировать, был ли синтаксический анализ формулы выполнен правильно.

Последним шагом в этом разделе станет встраивание парсера в электронную таблицу. Синтаксический анализ формулы является ответной реакцией на пользовательский ввод в ячейку. Полноценный ввод в ячейку смоделирован в Swing событием `TableUpdated`. Класс `TableUpdated` содержится в пакете `scala.swing.event`. Событие имеет следующую форму:

```
TableUpdated(table, rows, column)
```

Оно включает измененную таблицу, а также набор координат задействованных ячеек, заданный строками и столбцами. Параметр `rows` является значением диапазона типа `Range[Int]`[157] . Параметр `column` представляет собой целое число. Таким образом, в общем событие `TableUpdated` может

ссылаться на несколько задействованных ячеек, но они должны содержаться в последовательном диапазоне строк и относиться к одному и тому же столбцу.

Как только в таблицу внесены изменения, задействованные ячейки должны быть подвергнуты синтаксическому анализу заново. Для реагирования на событие `TableUpdated` нужно, как показано в листинге 35.6, добавить вариант к значению `reactions` компонента `table`.

Листинг 35.6. Электронная таблица, выполняющая синтаксический анализ формул

```
package org.stairwaybook.scells
import swing._
import event._

class Spreadsheet(val height: Int, val width: Int)
... {
  val table = new Table(height, width) {
    ...
    reactions += {
      case TableUpdated(table, rows, column) =>
        for (row <- rows)
          cells(row)(column).formula =
                FormulaParsers.parse(userData(row,
column))
    }
  }
}
```

Теперь при редактировании таблицы формулы всех задействованных ячеек будут обновляться за счет парсинга соответствующих пользовательских данных. После компиляции

рассмотренных на данный момент классов и запуска приложения `scells.Main` вы увидите такую же электронную таблицу, как та, что показана на рис. 35.3. Редактировать ячейки можно, набирая в них текст. После завершения редактирования в ячейке будет показана содержащаяся в ней формула. Можно также попробовать ввести в поле, имеющее фокус редактирования (см. рис. 35.3), какой-нибудь недопустимый текст, например `=add(1, X)`.

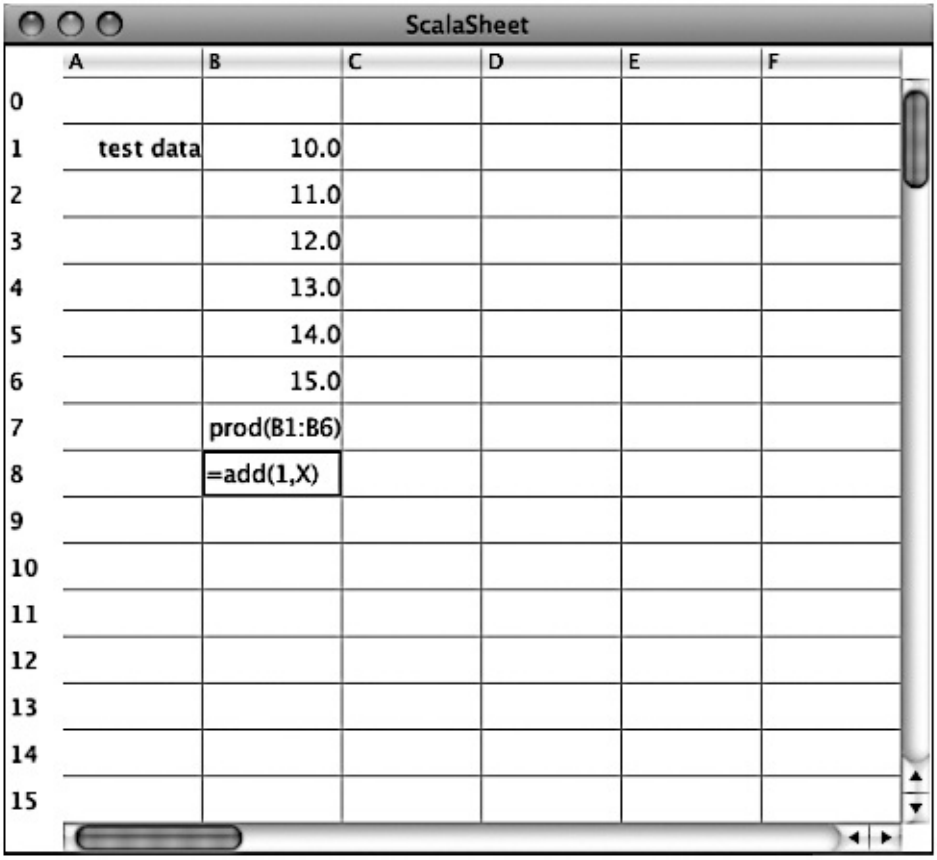


Рис. 35.3. Ячейки, показывающие свои формулы

О недопустимом вводе будет свидетельствовать сообщение об ошибке. К примеру, как только вы покинете редактируемое поле, показанное на рис. 35.3, в ячейке появится сообщение об ошибке `['(' expected]` (чтобы просмотреть сообщение об ошибке целиком, может понадобиться расширить столбец, перетаскивая вправо разделитель столбцов в заголовке).

35.5. Вычисление

Разумеется, в итоге электронная таблица должна не только показывать формулы, но и выполнять по ним вычисления. В этом разделе мы добавим все необходимые для этого компоненты.

Нам нужен метод `evaluate`, получающий формулу и возвращающий значение, представленное типом `Double`, которое вычисляется по этой формуле в текущей электронной таблице. Этот метод будет помещен в новый трейт по имени `Evaluator`. Методу требуется доступ к полям ячеек в классе `Model`, чтобы определить текущие значения ячеек, на которые имеются ссылки в формуле. В то же время классу `Model` нужно вызывать метод `evaluate`. Следовательно, между `Model` и `Evaluator` возникает взаимозависимость. Вполне подходящий способ выражения подобной взаимозависимости между классами был показан в главе 29: в одном направлении следует использовать наследование, а в другом — self-типы.

В примере электронной таблицы класс `Model` является наследником класса `Evaluator`, получая таким образом доступ к его методу вычисления. В обратном направлении в классе `Evaluator` в качестве его self-типа определяется класс `Model`:

```
package org.stairwaybook.scells
trait Evaluator { this: Model => ...
```

Таким образом, предполагается, что значением `this` внутри класса `Evaluator` станет `Model` и массив ячеек будет доступен при написании кода в виде либо `cells`, либо `this.cells`.

После создания подобной связи сконцентрируемся на определении содержимого класса `Evaluator`. В листинге 35.7 показана реализация метода `evaluate`. Как вы и ожидали, в методе содержится поиск по шаблону в отношении различных типов формул. Для координат `Coord(row, column)` он возвращает значение массива ячеек по заданным координатам. Для числа

Number(v) он возвращает значение v. Для текстовой надписи Textual(s) он возвращает нуль. И наконец, для применения функции Application(function, arguments) он вычисляет значения всех аргументов, извлекает объект функции, соответствующий ее имени, из таблицы operations и применяет эту функцию к значениям всех аргументов.

Листинг 35.7. Метод evaluate трейта Evaluator

```
def evaluate(e: Formula): Double = try {
  e match {
    case Coord(row, column) =>
      cells(row)(column).value
    case Number(v) =>
      v
    case Textual(_) =>
      0
    case Application(function, arguments) =>
      val argvals = arguments flatMap evalList
      operations(function)(argvals)
  }
} catch {
  case ex: Exception => Double.NaN
}
```

В таблице operations имена функций отображаются на объекты функций. Определение выглядит следующим образом:

```
type Op = List[Double] => Double
val operations = new
collection.mutable.HashMap[String, Op]
```

Как видно из данного определения, операции моделируются как

функции от списков значений к значениям. Тип `Op` вводит весьма удобный псевдоним для типа операции.

Для защиты от ошибок ввода вычисление в `evaluate` заключено в блок `try-catch`. При вычислении формулы ячейки может сложиться множество неблагоприятных обстоятельств: координаты могут выйти за пределы диапазона, имена функций могут быть не определены, у функций может оказаться неверное количество аргументов, арифметические операции могут быть неприемлемыми или вызывающими переполнение буфера. Реакция на любую из этих ошибок одна и та же — возвращение значения «не число». Возвращенное значение, `Double.NaN`, в соответствии со спецификацией IEEE является представлением для вычисления, которое не имеет представляемого значения в виде числа с плавающей точкой. Это может случиться, к примеру, по причине переполнения буфера или деления на нуль. Метод `evaluate`, показанный в листинге 35.7, выбирает возвращение такого же значения и для всех прочих видов ошибок. Преимущество этой схемы заключается в том, что в ней проще разобраться и для ее реализации не требуется слишком большой объем кода. Недостатком является то, что все виды ошибок оказываются объединены, что не позволяет пользователю электронной таблицы получить подробный ответ на вопрос о причине сбоя. При желании вы можете поэкспериментировать с созданием более конкретных способов представления ошибок в приложении SCells.

Вычисление аргументов отличается от вычисления формул верхнего уровня. Аргументы могут быть списком, а функции верхнего уровня — не могут. Например, выражение аргумента `A1:A3` в `sum(A1:A3)` возвращает значения ячеек `A1`, `A2`, `A3` в списке. Этот список затем передается операции `sum`. В выражениях аргументов можно также смешивать списки и отдельные значения, как, например, в операции `sum(A1:A3, 1.0, C7)`, которой будет возвращена сумма пяти значений. Для обработки аргументов,

которые могут вычисляться в списки, имеется еще одна функция, evalList, получающая формулу и возвращающая список значений:

```
private def evalList(e: Formula): List[Double] = e
match {
  case Range(_, _) => references(e) map (_.value)
  case _ => List(evaluate(e))
}
```

Если аргумент-формула передан методу evalList в виде диапазона, имеющего тип Range, возвращаемое значение будет представлять собой список, состоящий из значений всех ячеек, ссылки на которые попадают в этот диапазон. Для любой другой формулы результатом будет список, состоящий из одного значения, получившегося в результате вычисления по этой формуле. Ячейки, на которые ссылается формула, вычисляются третьей функцией, references. Ее определение выглядит следующим образом:

```
def references(e: Formula): List[Cell] = e match {
  case Coord(row, column) =>
    List(cells(row)(column))
  case Range(Coord(r1, c1), Coord(r2, c2)) =>
    for (row <- (r1 to r2).toList; column <- c1 to
c2)
    yield cells(row)(column)
  case Application(function, arguments) =>
    arguments flatMap references
  case _ =>
    List()
  }
} // окончание трейта Evaluator
```

Метод `references` пока имеет более общий характер, чем необходимо. Это выражается в том, что он вычисляет список ячеек, на которые ссылается формула любого сорта, а не только формула типа `Range`. Чуть позже обнаружится, что эта дополнительная функциональная возможность требуется для вычисления наборов ячеек, нуждающихся в обновлении. Тело метода представляет собой простой поиск по шаблону в отношении разновидностей формул. Для координат `Coord(row, column)` поиск возвращает одноэлементный список, содержащий ячейку, имеющую эти координаты. Для выражения диапазона `Range(coord1, coord2)` он возвращает все ячейки между двумя координатами, вычисляемые с помощью выражения `for`. Для применения функции `Application(function, arguments)` поиск возвращает ячейки, на которые имеются ссылки в каждом аргументе выражения, объединенные посредством функции `flatMap` в единый список. Для остальных двух типов формул, `Textual` и `Number`, он возвращает пустой список.

35.6. Библиотеки операций

В самом классе `Evaluator` не определены операции, которые могут выполняться в отношении ячеек, — его таблица операций изначально пуста. Замысел заключается в определении таких операций в других трейтах, которые затем подмешиваются в класс `Model`. Пример трейта, реализующего обычные арифметические операции, показан в листинге 35.8.

Листинг 35.8. Библиотека арифметических операций

```
package org.stairwaybook.scells
trait Arithmetic { this: Evaluator =>
  operations += (
    "add" -> { case List(x, y) => x + y },
```

```
      "sub" -> { case List(x, y) => x - y },
      "div" -> { case List(x, y) => x / y },
      "mul" -> { case List(x, y) => x * y },
      "mod" -> { case List(x, y) => x % y },
      "sum" -> { xs => (0.0 /: xs)(_ + _) },
      "prod" -> { xs => (1.0 /: xs)(_ * _) }
    )
  }
```

Примечательно, что у этого трейта нет экспортируемых элементов. Он всего лишь наполняет таблицу `operations` в ходе инициализации. Трейт получает доступ к таблице за счет использования self-типа, принадлежащего классу `Evaluator`, то есть применяет ту же технологию доступа к модели, которая применяется в классе `Arithmetic`.

Из семи операций, определенных трейтом `Arithmetic`, пять являются бинарными, а две получают произвольное количество аргументов. Все бинарные операции действуют по одной и той же схеме. Например, операция сложения `add` определена с помощью следующего выражения:

```
{ case List(x, y) => x + y }
```

То есть она ожидает список аргументов, содержащий два элемента, `x` и `y`, и возвращает сумму `x` и `y`. Если в списке аргументов содержатся не два аргумента, выдается исключение `MatchError`. Это соответствует общей философии «пусть будет сбой», которой придерживается модель вычисления, принятая в приложении SCell. Согласно этой философии неверный ввод в ходе выполнения программы ожидаемо приводит к выдаче исключения, которое затем отлавливается конструкцией `try-catch`, имеющейся внутри метода вычисления.

Последние две операции, `sum` и `prod`, получают список аргументов произвольной длины и вставляют бинарные операции

между идущими друг за другом элементами. То есть они являются экземплярами схемы правой свертки, которая выражена в классе `List` операцией `/:`. Например, чтобы выполнить сложение списка чисел `List(x, y, z)`, операция вычисляет `0 + x + y + z` . Первый операнд `0` становится результатом в том случае, если список пуст.

Эту библиотеку операций можно встроить в приложение электронной таблицы путем подмешивания трейта `Arithmetic` в класс `Model`:

```
package org.stairwaybook.scells

class Model(val height: Int, val width: Int)
    extends Evaluator with Arithmetic {

  case class Cell(row: Int, column: Int) {
    var formula: Formula = Empty
    def value = evaluate(formula)

    override def toString = formula match {
      case Textual(s) => s
      case _ => value.toString
    }
  }

   ... // все остальное, как и прежде
}
```

Еще одно изменение в классе `Model` касается способа отображения самих ячеек. В новой версии отображаемое значение ячейки зависит от ее формулы. Если формула представлена полем типа `Textual`, содержимое поля отображается буквально. Во всех остальных случаях формула вычисляется и отображается результат этого вычисления.

Если скомпилировать измененные трейты и классы и заново запустить программу `Main`, то получится уже что-то напоминающее настоящую электронную таблицу.

Пример показан на рис. 35.4. В ячейки можно вводить формулы и заставлять выполнять по ним вычисления. Например, как только будет закрыт фокус редактирования ячейки `C5` на рис. 35.4, в ней будет показано число `86.0`, являющееся результатом вычисления формулы `sum(C1:C4)`.

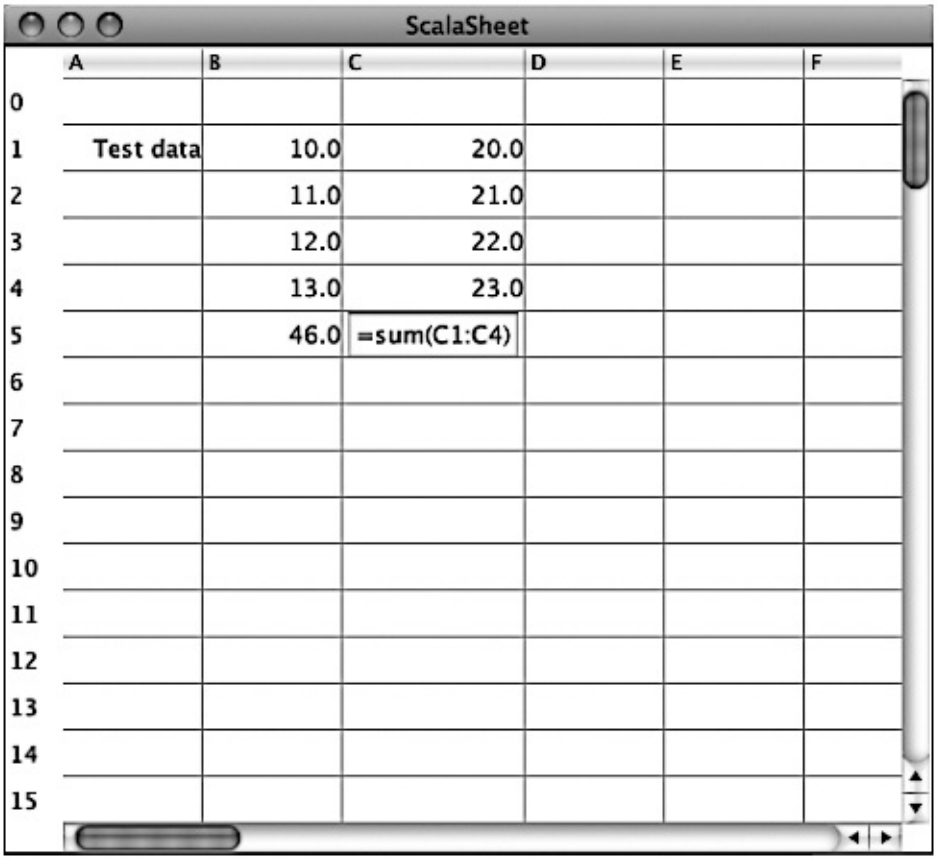


Рис. 35.4. Вычисляемые ячейки

Но в электронной таблице по-прежнему отсутствует весьма важная особенность. Если изменить значение ячейки `C1` с `20` на `100`, сумма в ячейке `C5` не будет автоматически обновлена до `166`. Для того чтобы увидеть изменения в `C5`, на этой ячейке придется щелкать вручную. У нас пока нет способа заставить ячейки после

изменения вычислять свои значения в автоматическом режиме.

35.7. Распространение изменений

Если значение ячейки изменилось, то все ячейки, значения которых от него зависят, должны заново вычислить и показать свои результаты. Проще всего добиться этого повторным вычислением каждой ячейки электронной таблицы после любого изменения. Но такой подход плохо поддается масштабированию в случае разрастания электронной таблицы.

Более удачным будет подход, когда заново вычисляются значения только тех ячеек, в формулах которых имеются ссылки на измененную ячейку. Он заключается в том, чтобы для распространения изменения воспользоваться средой публикаций-подписок на основе событий: как только ячейке присваивается формула, она будет подписываться на уведомления обо всех изменениях в тех ячейках, на которые ссылается эта формула. Значение, измененное в одной из таких ячеек, запустит перевычисление подписавшейся ячейки. Если такое повторное вычисление вызовет изменение в значении ячейки, она в свою очередь уведомит все ячейки, которые зависят от этого изменения. Процесс продолжится до тех пор, пока все значения ячеек не стабилизируются, то есть значения любых ячеек не прекратят изменяться[158].

Среда публикаций-подписок реализована в классе `Model` с использованием стандартного механизма событий имеющейся в Scala среды Swing. Новая (окончательная) версия этого класса выглядит следующим образом:

```
package org.stairwaybook.scells
import swing._

class Model(val height: Int, val width: Int)
```

```
extends Evaluator with Arithmetic {
```

По сравнению с предыдущей версией Model, в эту версию добавлен импорт swing._, позволяющий обращаться к абстракциям событий среды Swing напрямую. Основное изменение класса Model коснулось вложенного класса Cell. Теперь класс Cell является наследником класса Publisher, что позволяет ему публиковать события. Логика обработки событий целиком находится в set-методах двух свойств: value и formul. Новая версия Cell выглядит следующим образом:

```
case class Cell(row: Int, column: Int) extends
Publisher {
  override def toString = formula match {
    case Textual(s) => s
    case _ => value.toString
}
```

Это похоже на то, что value и formula являются двумя переменными в классе Cell. Они реализуются в двух закрытых полях, оснащенных открытыми get-методами value и formula и set-методами value_= и formula_=. Определение реализации свойства value имеет следующий вид:

```
private var v: Double = 0
def value: Double = v def value_=(w: Double) = {
  if (!(v == w || v.isNaN && w.isNaN)) {
    v = w     publish(ValueChanged(this))
  }
}
```

Set-метод value_= присваивает новое значение w закрытому полю v. Если новое значение отличается от старого, он также публикует событие ValueChanged с самой ячейкой в качестве аргумента. Следует заметить, что тест на изменение значения

немного усложнен, поскольку в нем используется значение NaN. В спецификации языка Java говорится, что NaN отличается от любого другого значения, включая самого себя! Поэтому равенство двух значений NaN рассматривается особым образом: значения v и w одинаковы, если они эквивалентны в отношении применения оператора == или оба являются значениями NaN, то есть оба выражения, v.isNaN и w.isNaN, выдают значение true.

Set-функция value_= выполняет в среде публикаций-подписок публикацию, в то время как set-функция formula_= осуществляет подписку:

```
private var f: Formula = Empty
def formula: Formula = f def formula_=(f: Formula)
= {
  for (c <- references(formula)) deafTo(c)
   this.f = f     for (c <- references(formula))
listenTo(c)
  value = evaluate(f)
}
```

Если ячейке присваивается новая формула, она сначала отписывается с помощью метода deafTo от событий во всех ячейках, на которые ссылалась прежняя формула. Затем в ней в закрытой переменной f сохраняется новая формула, а с помощью метода listenTo выполняется подписка на события во всех ячейках, на которые имеются ссылки в этой формуле. И наконец, значение данной ячейки вычисляется с использованием новой формулы.

В последнем фрагменте кода в переделанном классе Cell указывается, как следует реагировать на событие ValueChanged:

```
  reactions += {
            case ValueChanged(_) => value =
evaluate(formula)
```

```
  }
} // окончание класса Cell
```

Класс `ValueChanged` также содержится в классе `Model`:

```
case class ValueChanged(cell: Cell) extends
event.Event
```

Остальная часть класса `Model` остается неизменной:

```
  val cells = Array.ofDim[Cell](height, width)
  for (i <- 0 until height; j <- 0 until width)
    cells(i)(j) = new Cell(i, j)
} // окончание класса Model
```

Теперь разработка кода электронной таблицы почти завершена. Заключительный отсутствующий фрагмент относится к повторному отображению измененных ячеек. До сих пор все распространение значений касалось только внутренних `Cell`-значений, видимую таблицу они не затрагивали. Одним из способов изменения видимого представления может стать добавление к set-функции `value_=` команды перерисовки `redraw`. Но тогда нарушится наблюдавшееся до сих пор четкое разделение между моделью и представлением. Более модульный характер будет у решения об уведомлении таблицы обо всех событиях `ValueChanged` и предоставления ей возможности выполнять перерисовку самостоятельно. Последний компонент электронной таблицы, реализующий эту схему, показан в листинге 35.9.

Листинг 35.9. Завершающий компонент электронной таблицы

```
package org.stairwaybook.scells
import swing._, event._
```

```
class Spreadsheet(val height: Int, val width: Int)
    extends ScrollPane {
  val cellModel = new Model(height, width)
  import cellModel._

  val table = new Table(height, width) {
    ... // все установки, как в листинге 35.1

    override def rendererComponent(
        isSelected: Boolean, hasFocus: Boolean,
        row: Int, column: Int) =
      ... // как в листинге 35.3
    def userData(row: Int, column: Int): String =
      ... // как в листинге 35.3

    reactions += {
      case TableUpdated(table, rows, column) =>
        for (row <- rows)
          cells(row)(column).formula =
                FormulaParsers.parse(userData(row,
column))
      case ValueChanged(cell) =>
        updateCell(cell.row, cell.column)
    }

    for (row <- cells; cell <- row) listenTo(cell)
  }
  val rowHeader = new ListView(0 until height) {
    fixedCellWidth = 30
    fixedCellHeight = table.rowHeight
  }
  viewportView = table
```

```
    rowHeaderView = rowHeader
  }
```

Класс `Spreadsheet`, показанный в листинге 35.9, содержит только два пересмотренных фрагмента. Во-первых, теперь компонент `table` с помощью метода `listenTo` подписан на события, происходящие во всех ячейках модели. Во-вторых, в реакциях таблицы появился новый case-вариант: если поступает уведомление о событии `ValueChanged(cell)`, выдается запрос на перерисовку соответствующей ячейки вызовом метода `updateCell(cell.row, cell.column)`.

Резюме

Разработанная в этой главе электронная таблица не имеет функциональных ограничений, хотя в некоторых местах использованы не самые удобные для пользователя, но самые простые варианты реализации. Это позволило уместить код менее чем в 200 строках. Несмотря на это, архитектура электронной таблицы поддерживает возможность ее изменения и расширения. Если захочется продолжить эксперименты с кодом, то в качестве возможных изменений и добавлений предлагается следующее.

Сделать электронную таблицу изменяемой в размерах, чтобы количество строк и столбцов корректировалось в интерактивном режиме.

48. Добавить новые виды формул, например бинарные операции или другие функции.

49. Подумать над тем, что делать, когда ячейки рекурсивно ссылаются друг на друга. Например, если ячейка `A1` содержит формулу `add(B1, 1)`, а ячейка `B1` — формулу `mul(A1, 2)`, повторное вычисление любой ячейки приведет к переполнению стека. Понятно, что ничего хорошего в таком решении нет. В

качестве альтернативных решений можно либо не допустить подобной ситуации, либо просто выполнять только одну итерацию при каждом задействовании одной из ячеек.

50. Усовершенствовать систему обработки ошибок, выдавая более подробные сообщения, описывающие причины сбоев.

51. Добавить в верхнюю часть таблицы поле ввода формул, чтобы было удобнее вводить длинные формулы.

В начале книги подчеркивался аспект масштабируемости программ на Scala. Утверждалось, что сочетание допускаемых в Scala объектно-ориентированных и функциональных конструкций делает язык подходящим для программ в диапазоне от небольших сценариев до очень крупных систем. Очевидно, что представленная здесь электронная таблица относится к небольшим системам, даже если учесть, что при ее создании с использованием большинства других языков программирования количество строк кода наверняка было бы намного больше 200. Несмотря на это, при работе над этим приложением вам представилась возможность ознакомиться со многими подробностями языка Scala, позволяющими расширять масштабы приложений.

В электронной таблице использовались классы и трейты Scala с возможностью весьма гибко объединять компоненты. Рекурсивные зависимости между компонентами выражались с помощью self-типов. Потребность в статическом состоянии была полностью исключена, единственными компонентами верхнего уровня, не являющимися классами, были деревья формул и парсеры формул, и все они имели чисто функциональный характер. В приложении широко использовались также функции высшего порядка и поиск по шаблону, которые применялись как при обращении к формулам, так и при обработке событий. Таким образом, у нас неплохо получилось продемонстрировать, как легко можно объединить функциональное и объектно-ориентированное

программирование.

Одной из причин такой лаконичности приложения электронной таблицы является его опора на весьма мощные библиотеки. Библиотека парсер-комбинаторов, по сути, предоставила внутренний предметно-ориентированный язык написания парсеров. Без него было бы гораздо сложнее выполнить синтаксический анализ формул. Обработка событий в имеющейся в Scala библиотеке Swing стала хорошим примером эффективности управляющих абстракций. Если вам знакомы библиотеки Swing, входящие в Java, то вы, вероятно, сможете по достоинству оценить краткость концепции реакций в Scala, особенно по сравнению с рутиной создания методов уведомления и реализации интерфейсов отслеживания в классическом шаблоне проектирования публикаций-подписок. Таким образом, электронная таблица позволила продемонстрировать преимущества расширяемости, при которой библиотеки верхнего уровня могут быть очень похожими на расширения самого языка.

156 Хотя код this(row, column) может быть похож на вызов конструктора, в данном случае он является вызовом метода apply текущего экземпляра класса Table.

157 Range[Int] также является типом такого выражения Scala, как 1 to N.

158 Здесь предполагается, что между ячейками нет циклических зависимостей. Противоположное будет рассмотрено в конце главы.

Приложение. Сценарии Scala на Unix и Windows

В Unix можно запустить сценарий на Scala в качестве сценария оболочки, предваряя этот запуск «шебанг»-инструкцией в самом начале файла. Наберите, к примеру, для файла `helloarg` следующий код:

```
#!/bin/sh
exec scala "$0" "$@"
!#
// Поприветствуйте первый аргумент
println("Hello, " + args(0) + "!")
```

Начальная команда `#!/bin/sh` должна быть самой первой строкой файла. Как только будет установлено разрешение на выполнение:

```
$ chmod +x helloarg
```

можно будет запустить сценарий на Scala в качестве сценария оболочки простой командой:

```
$ ./helloarg globe
```

В Windows такой же эффект можно получить, назвав файл `helloarg.bat` и поместив в верхней части своего сценария следующий код:

```
::#!
@echo off
call scala %0 % *
goto :eof
::!#
```