С примерами на TypeScript

Программируй & типизируй

Влад Ришкуция

MANNING

Programming with Types

WITH EXAMPLES IN TYPESCRIPT

VLAD RISCUTIA

MANNING
SHELTER ISLAND

Влад Ришкуция

Программируй & типизируй

ПИТЕР® Санкт-Петербург • Москва • Минск 2021

ББК 32.973.2-018
УДК 004.42
Р57

Ришкуция Влад

Р57 Программируй & типизируй. — СПб.: Питер, 2021. — 352 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1692-8

Причиной многих программных ошибок становится несоответствие типов данных. Сильная система типов позволяет избежать целого класса ошибок и обеспечить целостность данных в рамках всего приложения. Разработчик, научившись мастерски использовать типы в повседневной практике, будет создавать более качественный код, а также сэкономит время, которое потребовалось бы для выискивания каверзных ошибок, связанных с данными.

В книге рассказывается, как с помощью типизации создавать программное обеспечение, которое не только было бы безопасным и работало без сбоев, но также обеспечивало простоту в сопровождении.

Примеры решения задач, написанные на TypeScript, помогут развить ваши навыки работы с типами, начиная от простых типов данных и заканчивая более сложными понятиями, такими как функторы и монады.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018
УДК 004.42

ISBN 978-1617296413 англ.
ISBN 978-5-4461-1692-8

Краткое содержание

Оглавление

Моей жене Диане за ее безграничное терпение.

Предисловие

Эта книга — итог многих лет изучения систем типов и правильности работы программного обеспечения, выраженный в виде практического руководства по созданию реальных приложений.

Мне всегда нравилось искать способы написания более совершенного кода, но собственно начало этой книги, мне кажется, было заложено в 2015 году. Я тогда перешел из одной команды разработчиков в другую и хотел обновить свои знания языка C++. Я начал смотреть видео с конференций по C++, читать книги Александра Степанова по обобщенному программированию и полностью изменил свои представления о том, как нужно писать код.

Параллельно в свободное время я изучал Haskell и осваивал продвинутые свойства его системы типов. Программируя на функциональном языке, начинаешь понимать, как много естественных возможностей подобных языков со временем приживаются в более распространенных языках.

Я прочитал немало книг на эту тему, начиная от *Elements of Programming* и *From Mathematics to Generic Programming* Степанова[1] и до *Category Theory for Programmers* Бартоша Милевски (Bartosz Milewski)[2] и *Types and Programming Languages* Бенджамина Пирса (Benjamin Pierce)[3]. Как вы понимаете из названий, книги посвящены скорее теоретико-математическим вопросам. Чем больше я узнавал о системах типов, тем лучше становился код, который я писал на работе. Между теоретическими вопросами проектирования систем типов и повседневной работой над программным обеспечением существует самая непосредственная связь. Я вовсе не открываю Америку: все причудливые возможности систем типов существуют как раз для решения реальных задач.

---

[1] *Степанов А., Мак-Джонс П.* Начала программирования. — М.: Вильямс, 2011. *Роуз Д., Степанов А. А.* От математики к обобщенному программированию. — М.: ДМК Пресс, 2015.

[2] «Теория категорий для программистов». Ее неофициальный перевод можно найти на сайте https://henrychern.wordpress.com/2017/07/17/httpsbartoszmilewski-com20141028category-theory-for-programmers-the-preface/. — *Примеч. пер.*

[3] *Пирс Б.* Типы в языках программирования. — М.: Лямбда пресс; Добросвет, 2011.

Я осознал, что далеко не у всех практикующих программистов есть время и желание читать объемные книги с математическими доказательствами. С другой стороны, я не потратил время впустую за чтением этих книг: благодаря им я стал лучшим специалистом по программному обеспечению. Мне стало понятно, что есть потребность в книге, в которой бы описывались системы типов и их преимущества на менее формальном языке, с упором на практическое применение в ежедневной работе.

Цель книги — подробный анализ возможностей систем типов, начиная от базовых типов, функциональных типов и создания подтипов[1], ООП, обобщенного программирования и типов более высокого рода, например функторов и монад. Вместо того чтобы сосредоточиться на теоретической стороне этих возможностей, я опишу их практическое применение. В данной книге рассказывается, как и когда использовать каждую из них, чтобы сделать свой код лучше.

Изначально предполагалось, что примеры кода будут на языке C++. Система типов C++ обладает намного бо́льшими возможностями, чем у таких языков, как Java и C#. С другой стороны, C++ — сложный язык, и я не хотел искусственно ограничивать аудиторию книги, так что решил применить вместо него TypeScript. Система типов этого языка тоже располагает широкими возможностями, но его синтаксис более доступен, поэтому изучение примеров не доставит сложностей даже тем, кто привык к другим языкам. В приложении Б приведена краткая шпаргалка по используемому в данной книге подмножеству TypeScript.

Я надеюсь, что вы получите удовольствие от чтения этой книги и изучите кое-какие новые методики, которые сможете сразу же применить в своих проектах.

[1] Здесь и далее для единообразия subtype/supertype переводится как «подтип/надтип», хотя в русскоязычной литературе первое чаще называют «подтип» (а не субтип), а второе — «супертип». — *Примеч. пер.*

Благодарности

Прежде всего я хотел бы поблагодарить мою семью за поддержку и понимание. На каждом этапе данного пути со мной были моя жена Диана и дочь Ада, поддерживая меня и предоставляя свободу, необходимую для завершения этой книги.

Написание книги, безусловно, заслуга целой команды. Я признателен Майклу Стивенсу (Michael Stephens) за первоначальные отзывы. Я хочу поблагодарить моего редактора Элешу Хайд (Elesha Hyde) за всю ее помощь, советы и отзывы. Спасибо Майку Шепарду (Mike Shepard) за рецензию на каждую из глав и честную критику. Кроме того, спасибо Херману Гонсалесу (German Gonzales) за просмотр всех до единого примеров кода и проверку правильности их работы. Я хотел бы поблагодарить всех рецензентов за уделенное мне время и бесценные отзывы. Спасибо вам, Виктор Бек (Viktor Bek), Роберто Касадеи (Roberto Casadei), Ахмед Чиктэй (Ahmed Chicktay), Джон Корли (John Corley), Джастин Коулстон (Justin Coulston), Тео Деспудис (Theo Despoudis), Дэвид Ди Мария (David DiMaria), Кристофер Фрай (Christopher Fry), Херман Гонсалес-Моррис (German Gonzalez-Morris), Випул Гупта (Vipul Gupta), Питер Хэмптон (Peter Hampton), Клайв Харбер (Clive Harber), Фред Хит (Fred Heath), Райан Хьюбер (Ryan Huber), Дес Хорсли (Des Horsley), Кевин Норман Д. Капчан (Kevin Norman D. Kapchan), Хосе Сан-Леандро (Jose San Leandro), Джеймс Люй (James Liu), Уэйн Мазер (Wayne Mather), Арнальдо Габриэль Айала Мейер (Arnaldo Gabriel Ayala Meyer), Риккардо Новьелло (Riccardo Noviello), Марко Пероне (Marco Perone), Джермаль Прествуд (Jermal Prestwood), Борха Кеведо (Borja Quevedo), Доминго Себастьян Састре (Domingo Sebastián Sastre), Рохит Шарм (Rohit Sharm) и Грег Райт (Greg Wright).

Я хотел бы поблагодарить моих сослуживцев и наставников за все, чему они меня научили. Когда я изучал возможности применения типов для улучшения нашей кодовой базы, мне повезло встретить нескольких замечательных менеджеров, всегда готовых прийти на помощь. Спасибо Майку Наварро (Mike Navarro), Дэвиду Хансену (David Hansen) и Бену Россу (Ben Ross) за их веру в меня.

Спасибо всему сообществу разработчиков C++, от которых я столь многому научился, особенно Шону Пэренту (Sean Parent) — за вдохновение и замечательные советы.

О книге

Цель этой книги — продемонстрировать вам, как писать лучший, более безопасный код с помощью систем типов. Хотя большинство изданий, посвященных системам типов, сосредотачиваются на более формальных аспектах вопроса, данная книга представляет собой скорее практическое руководство. Она содержит множество примеров, приложений и сценариев, встречающихся в повседневной работе программиста.

Целевая аудитория

Книга предназначена для программистов-практиков, которые хотят узнать больше о функционировании систем типов и о том, как с их помощью повысить качество своего кода. Желательно иметь опыт работы с объектно-ориентированными языками программирования: Java, C#, C++ или JavaScript/TypeScript, а также хотя бы минимальный опыт проектирования ПО. Хотя в этой книге рассматриваются различные методики написания надежного, пригодного для компоновки и хорошо инкапсулированного кода, предполагается, что вы понимаете, почему эти свойства желательны.

Структура книги

Книга содержит 11 глав, посвященных различным аспектам типизированного программирования.

- В главе 1 мы познакомимся с типами и их системами, обсудим, для чего они служат и какую пользу могут принести. Рассмотрим существующие виды систем типов и поговорим о строгой, а также статической и динамической типизациях.
- В главе 2 мы рассмотрим простые типы данных, существующие в большинстве языков программирования, и нюансы, которые следует учитывать при их

использовании. К распространенным простым типам данных относятся: пустой тип и единичный тип, булевы значения, числа, строки, массивы и ссылки.

- В главе 3 мы изучим сочетаемость: разнообразные способы сочетания типов для определения новых типов. Кроме того, рассмотрим различные способы реализации паттерна проектирования «Посетитель» и алгебраические типы данных.
- В главе 4 мы поговорим о типобезопасности — пути снижения неоднозначности и предотвращения ошибок при использовании типов. Кроме того, я расскажу о добавлении/удалении информации о типе из кода с помощью приведения типов.
- В главе 5 вы познакомитесь с функциональными типами и возможностями, возникающими благодаря созданию функциональных переменных. Мы рассмотрим альтернативные способы реализации паттерна проектирования «Стратегия» и конечных автоматов, а также базовые алгоритмы `map()`, `filter()` и `reduce()`.
- В главе 6 будет представлен расширенный материал предыдущей главы и продемонстрировано несколько продвинутых приложений функциональных типов данных, начиная от упрощенного паттерна проектирования «Декоратор» и заканчивая возобновляемыми и асинхронными функциями.
- В главе 7 мы познакомимся с созданием подтипов и обсудим совместимость типов данных. Мы рассмотрим применение низшего и высшего типов и увидим, как связаны друг с другом тип-сумма, коллекции и функциональные типы с точки зрения подтипизации.
- В главе 8 мы обсудим ключевые элементы объектно-ориентированного программирования и их использование. Рассмотрим интерфейсы, наследование, сочетание типов и примеси.
- В главе 9 вы познакомитесь с обобщенным программированием и его первым приложением: обобщенными структурами данных. Эти структуры отделяют схему данных от них самих; обход структур возможен с помощью итераторов.
- В главе 10 мы продолжим тему обобщенного программирования и обсудим обобщенные алгоритмы и категории итераторов. Обобщенными являются алгоритмы, которые можно использовать повторно для различных типов данных. Итераторы играют роль интерфейса между структурами данных и алгоритмами и, в зависимости от своих возможностей, могут запускать различные алгоритмы.
- В главе 11, заключительной, будут описаны типы, принадлежащие к более высокому роду, и дано объяснение, что такое функторы и монады и как их использовать. Завершат эту главу ссылки на литературу для дальнейшего изучения.

Все главы используют понятия, описанные в предыдущих главах книги, так что читать их следует по порядку. Тем не менее четыре основные темы более или менее независимы. В первых четырех главах описываются основные понятия; в главах 5 и 6 рассказывается о функциональных типах данных; в главах 7 и 8 — о создании подтипов; главы 9, 10 и 11 посвящены обобщенному программированию.

О коде

Эта книга содержит множество примеров исходного кода как в пронумерованных листингах, так и внутри обычного текста. В обоих случаях исходный код набран `вот таким моноширинным шрифтом` с целью отличить его от обычного текста. Иногда код также набран **`полужирным шрифтом`**, чтобы подчеркнуть изменения по сравнению с предыдущими шагами в текущей главе, например при добавлении новой функциональной возможности к уже существующей строке кода.

Во многих случаях первоначальный исходный код был переформатирован: были добавлены разрывы строк и переработаны отступы, чтобы наилучшим образом использовать доступное место на страницах книги. В редких случаях этого оказывалось недостаточно, и листинги включают маркеры продолжения строки (➥). Кроме того, из исходного кода нередко удалялись комментарии там, где он описывался в тексте. Многие листинги сопровождают примечания к коду, подчеркивающие важные нюансы.

Исходный код для всех листингов данной книги доступен для скачивания с GitHub по адресу https://github.com/vladris/programming-with-types/. Сборка кода производилась с помощью версии 3.3 компилятора TypeScript со значением ES6 для опции `target` и с опцией `strict`.

Об авторе

Влад Ришкуция — специалист по разработке ПО в Microsoft, имеет более чем десятилетний опыт. За это время он руководил несколькими крупными программными проектами и обучил множество молодых специалистов.

Дискуссионный форум книги

Покупка этой книги дает право на бесплатный доступ к частному веб-форуму издательства Manning, где можно оставлять комментарии о книге, задавать технические вопросы и получать помощь от автора книги и других пользователей. Чтобы попасть на этот форум, перейдите по адресу https://livebook.manning.com/#!/book/natural-language-processing-in-action/discussion. Узнать больше о форумах издательства и правилах поведения на них можно на странице https://livebook.manning.com/#!/discussion.

Обязательства издательства Manning по отношению к своим читателям заключаются в том, чтобы предоставить место для содержательного диалога между отдельными читателями, а также читателями и авторами. Эти обязательства не включают какого-либо конкретного объема участия со стороны авторов, чей вклад в работу форума остается добровольным (и неоплачиваемым). Мы советуем вам задавать авторам интересные и трудные вопросы, чтобы их интерес не угас!

Об иллюстрации на обложке

Рисунок на обложке называется *Fille Lipporolle en habit de Noce* («Девица Липороль в свадебном платье»). Эта иллюстрация взята из недавнего переиздания книги Жака Грассе де Сан-Савье *Costumes de Différents Pays* («Наряды разных стран»), опубликованной во Франции в 1797 году. Все иллюстрации прекрасно прорисованы и раскрашены вручную. Широкое разнообразие коллекции нарядов Грассе де Сан-Савье напоминает нам, насколько разъединены были различные регионы мира всего 200 лет назад. Изолированные друг от друга, люди говорили на разных диалектах и языках. На улицах городов и в деревнях по одной манере одеваться можно было легко понять, каким ремеслом занимается человек и каково его социальное положение.

Стили одежды с тех пор изменились, и столь богатое разнообразие различных регионов угасло. Зачастую непросто отличить даже жителя одного континента от жителя другого, не говоря уже о городах и странах. Возможно, мы пожертвовали культурным многообразием в пользу большего разнообразия личной жизни — и определенно более разнообразной и динамичной жизни технологической.

В наше время, когда книги на компьютерную тематику так мало отличаются друг от друга, издательство Manning отдает должное изобретательности и инициативе компьютерного бизнеса обложками книг, основанными на богатом разнообразии жизни в разных уголках мира двухвековой давности, возвращенном к нам иллюстрациями Жака Грассе де Сан-Савье.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1 Введение в типизацию

В этой главе

- Зачем нужны системы типов.
- Преимущества сильно типизированного кода.
- Разновидности систем типов.
- Распространенные возможности систем типов.

Аппарат Mars Climate Orbiter развалился в атмосфере Марса, поскольку разработанный компанией Lockheed компонент выдавал измерения импульса силы в фунт-силах на секунду (единицы измерения США), а другой компонент, разработанный НАСА, ожидал, что импульс силы будет измеряться в ньютонах на секунду (единицы СИ). Катастрофы можно было избежать, если бы для этих двух величин использовались различные типы данных.

Как мы будем наблюдать на протяжении данной книги, проверки типов позволяют исключать целые классы ошибок при условии наличия достаточной информации. По мере роста сложности программного обеспечения должны обеспечиваться и лучшие гарантии правильности его работы. Мониторинг и тестирование могут продемонстрировать, ведет ли себя ПО в соответствии со спецификациями в заданный момент времени при определенных входных данных. Типы же обеспечивают более общее подтверждение должного поведения кода, независимо от входных данных.

Благодаря научным изысканиям в области языков программирования возникают все более и более эффективные системы типов (см., например, такие языки

программирования, как Elm и Idris). Растет популярность языка Haskell. В то же время продолжаются попытки добиться проверки типов на стадии компиляции в динамически типизированных языках: в Python появилась поддержка указаний ожидаемых типов (type hints) и был создан язык TypeScript, единственная цель которого — обеспечить проверку типов во время компиляции в JavaScript.

Типизация кода, безусловно, важна, и благодаря полному использованию возможностей системы типов, предоставляемой языком программирования, можно писать лучший, более безопасный код.

1.1. Для кого эта книга

Книга предназначена для программистов-практиков. Читатель должен хорошо уметь писать код на одном из таких языков программирования, как Java, C#, C++ или JavaScript/TypeScript. Примеры кода приведены на языке TypeScript, но большая часть излагаемого материала применима к любому языку программирования. На самом деле в примерах далеко не всегда используется характерный TypeScript. По возможности они адаптировались так, чтобы их понимали программисты на других языках программирования. Сборка примеров кода описана в приложении А, а краткая «шпаргалка» по языку TypeScript — в приложении Б.

Если вы по работе занимаетесь разработкой объектно-ориентированного кода, то, возможно, слышали об алгебраических типах данных (algebraic data type, ADT), лямбда-выражениях, обобщенных типах данных (generics), функторах, монадах и хотите лучше разобраться, что это такое и как их использовать в своей работе.

Эта книга расскажет, как использовать систему типов языка программирования для проектирования менее подверженного ошибкам, более модульного и понятного кода. Вы увидите, как превратить ошибки времени выполнения, которые могут привести к отказу всей системы, в ошибки компиляции и перехватить их, пока они еще не натворили бед.

Основная часть литературы по системам типов сильно формализована. Книга же сосредотачивает внимание на практических приложениях систем типов; поэтому математики в ней очень мало. Тем не менее желательно, чтобы вы имели представление об основных понятиях алгебры, таких как функции и множества. Это понадобится для пояснения некоторых из нужных нам понятий.

1.2. Для чего существуют типы

На низком уровне аппаратного обеспечения и машинного кода логика программы (код) и данные, которыми она оперирует, представлены в виде битов. На этом уровне нет разницы между кодом и данными, так что вполне могут возникнуть ошибки, при которых система путает одно с другим. Их диапазон простирается от фатальных сбоев программы до серьезных уязвимостей, когда злоумышленник обманом заставляет систему считать входные данные кодом, подлежащим выполнению.

Пример подобной нестрогой интерпретации — функция `eval()` языка JavaScript, выполняющая строковое значение как код. Она отлично работает, если переданная ей строка представляет собой допустимый код на языке JavaScript, но вызывает ошибку времени выполнения в противном случае, как показано в листинге 1.1.

Листинг 1.1. Попытка интерпретировать данные как код

```
console.log(eval("40+2"));
console.log(eval("Hello world!"));
```

Выводит в консоль 42

Порождает исключение SyntaxError: unexpected token: identifier

1.2.1. Нули и единицы

Необходимо не только различать код и данные, но и интерпретировать элементы данных. Состоящая из 16 бит последовательность `1100001010100011` может соответствовать беззнаковому 16-битному целому числу `49827`, 16-битному целому числу со знаком `-15709`, символу `'£'` в кодировке UTF-8 или чему-то совершенно другому, как можно видеть на рис. 1.1. Аппаратное обеспечение, на котором работают наши программы, хранит все в виде последовательностей битов, так что необходим дополнительный слой для осмысления этих данных.

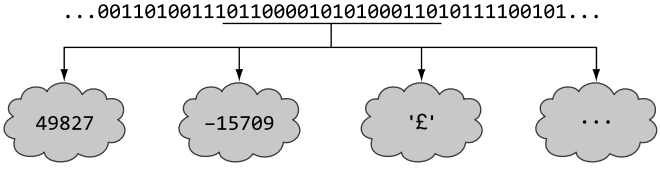


Рис. 1.1. Последовательность битов можно интерпретировать по-разному

Типы придают смысл подобным данным и указывают программному обеспечению, как интерпретировать заданную последовательность битов в установленном контексте, чтобы она сохранила задуманный автором смысл.

Кроме того, типы ограничивают множество допустимых значений переменных. Шестнадцатибитное целое число со знаком может отражать любое из целочисленных значений от `-32768` до `32767` и только их. Благодаря ограничению диапазона допустимых значений исключаются целые классы ошибок, поскольку не допускается возникновения неправильных значений во время выполнения, как показано на рис. 1.2. Чтобы понять многие из приведенных в этой книге концепций, важно рассматривать типы как множества возможных значений.

В разделе 1.3 мы увидим: система обеспечивает также соблюдение многих других мер безопасности при добавлении возможностей в код, например обозначение значений как `const` или членов как `private`.

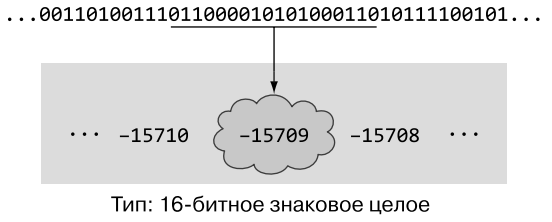


Рис. 1.2. Последовательность битов с типом 16-битного знакового целого. Информация о типе (16-битное знаковое целое число) указывает компилятору и/или среде выполнения, что эта битная последовательность представляет собой целочисленное значение в диапазоне от –32 768 до 32 767, благодаря чему она правильно интерпретируется как –15 709

1.2.2. Что такое типы и их системы

Раз уж книга посвящена типам и их системам, дам определения этих терминов, прежде чем идти дальше.

ЧТО ТАКОЕ ТИП

Тип (type) — классификация данных, определяющая допустимые операции над ними, смысл этих данных и множество допустимых значений. Компилятор и/или среда выполнения производят проверку типов, чтобы обеспечить целостность данных и соблюдение ограничений доступа, а также интерпретацию данных в соответствии с замыслом разработчика.

В некоторых случаях ради простоты мы будем игнорировать относящуюся к операциям часть этого определения и рассматривать типы просто как множества, отражающие все возможные значения экземпляра данного типа.

СИСТЕМА ТИПОВ

Система типов (type system) представляет собой набор правил присвоения типов элементам языка программирования и обеспечения соблюдения этих присвоений. Такими элементами могут быть переменные, функции и другие высокоуровневые конструкции языка. Системы типов производят присвоение типов с помощью задаваемой в коде нотации или неявным образом, путем вывода типа конкретного элемента по контексту. Системы типов разрешают одни преобразования типов друг в друга и запрещают другие.

Теперь, когда мы узнали определения типов и систем типов, посмотрим, как обеспечивается соблюдение правил системы типов. На рис. 1.3 показано на высоком уровне выполнение исходного кода.

Если описывать на очень высоком уровне, то создаваемый нами исходный код преобразуется компилятором или интерпретатором в инструкции для машины *(среды выполнения)*. Ее роль может играть физическая машина (в этом случае роль инструкций играют инструкции CPU) или виртуальная с собственным набором инструкций и функций.

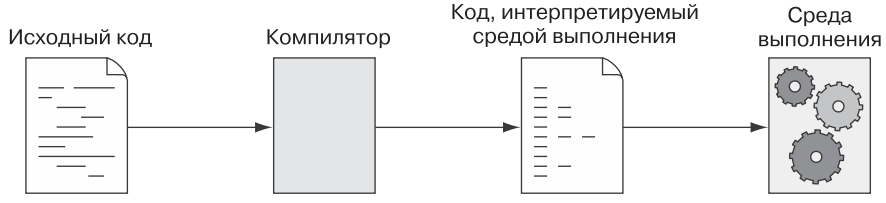


Рис. 1.3. С помощью компилятора или интерпретатора исходный код преобразуется в код, запускаемый средой выполнения. Ее роль может играть физический компьютер или виртуальная машина, например JVM Java или движок JavaScript браузера

ПРОВЕРКА ТИПОВ

Процесс проверки типов (type checking) обеспечивает соблюдение программой правил системы типов. Проверка производится компилятором во время преобразования кода или средой выполнения при его работе. Компонент компилятора, обеспечивающий соблюдение правил типизации, называется модулем проверки типов (type checker).

Если проверка типов завершается неудачно, то есть программа не соблюдает правила системы типов, то возникает ошибка на этапе компиляции или выполнения. Разницу между проверкой типа на этапе компиляции и на этапе выполнения мы обсудим подробнее в разделе 1.4.

Проверка типов и доказательства

В основе систем типов лежит формальная теория. Замечательное соответствие Карри-Ховарда (Curry-Howard correspondence), известное также как эквивалентность между математическими доказательствами и программами (proofs-as-programs), демонстрирует родственность логики и теории типов. Оно показывает, что тип можно рассматривать как логическое высказывание, а функцию, принимающую на входе один тип и возвращающую другой, — как логическую импликацию. Значение типа эквивалентно факту справедливости высказывания.

Возьмем для примера функцию, принимающую на входе `boolean` и возвращающую `string`.

Из булева значения в строковое

```
function booleanToString(b: boolean): string {
    if (b) {
        return "true";
    } else {
        return "false";
    }
}
```

Эту функцию можно интерпретировать как «из `boolean` следует `string`». По заданному факту высказывания типа `boolean` данная функция (импликация) выдает факт

высказывания типа `string`. Факт `boolean` представляет собой значение этого типа, `true` или `false`. По нему указанная функция (импликация) выдает факт `string` в виде строки `"true"` или `"false"`.

Тесная связь между логикой и теорией типов показывает: соблюдающая правила системы типов программа эквивалентна логическому доказательству. Другими словами, система типов — язык написания этих доказательств. Соответствие Карри-Ховарда важно тем, что правильность работы программы гарантируется с логической строгостью.

1.3. Преимущества систем типов

Все данные, по сути, представляют собой нули и единицы, поэтому все свойства данных, например их интерпретация, неизменяемость и видимость, относятся к уровню типа. Переменная объявляется с числовым типом, и модуль проверки типа гарантирует, что ее значение не будет интерпретировано как строковое. Переменная объявляется как приватная или предназначенная только для чтения. И хотя сами данные в памяти ничем не отличаются от аналогичных публичных изменяемых данных, модуль проверки типа гарантирует, что мы не будем обращаться к приватной переменной вне ее области видимости или пытаться изменить данные, предназначенные только для чтения.

Основные преимущества типизации — *корректность* (correctness), *неизменяемость* (immutability), *инкапсуляция* (encapsulation), *компонуемость* (composability) и *читабельность* (readability). Это фундаментальные признаки хорошей архитектуры и нормального поведения программного обеспечения. С течением времени системы развиваются. Эти признаки противостоят энтропии, которая неизбежно возникает в любой системе.

1.3.1. Корректность

Корректным (correct) является код, который ведет себя в соответствии со спецификациями, выдает ожидаемые результаты без ошибок и сбоев во время выполнения. Благодаря типам растут строгость кода и гарантии его должного поведения.

Для примера предположим, что нам нужно найти позицию строки `"Script"` внутри другой строки. Мы не будем передавать достаточную информацию о типе и разрешим передачу в качестве аргумента нашей функции значения типа `any`. Как показывает листинг 1.2, это приведет к ошибкам во время выполнения.

В этой программе содержится ошибка — `42` не является допустимым аргументом для функции `scriptAt`, но компилятор об этом молчит, поскольку мы не предоставили достаточную информацию о типе данных. Усовершенствуем данный код, ограничив аргумент типом `string` в листинге 1.3.

Теперь компилятор отвергает эту некорректную программу, выдавая следующее сообщение об ошибке: `Argument of type '42' is not assignable to parameter of type 'string'` (невозможно присвоить параметру типа `'string'` аргумент типа `'42'`).

Листинг 1.2. Недостаточная информация о типе данных

```
function scriptAt(s: any): number {
    return s.indexOf("Script");
}

console.log(scriptAt("TypeScript"));
console.log(scriptAt(42));
```

Тип аргумента s — any, то есть разрешается значение произвольного типа

Эта строка выводит в консоль корректное значение 4

Передача в качестве аргумента числового значения приводит к TypeError во время выполнения

Листинг 1.3. Уточненная информация о типе

```
function scriptAt(s: string): number {
    return s.indexOf("Script");
}

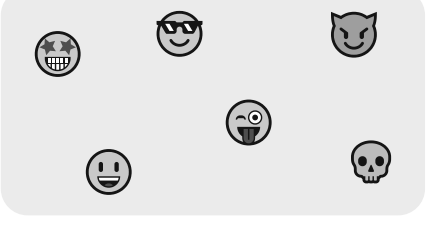
console.log(scriptAt("TypeScript"));
console.log(scriptAt(42));
```

Теперь у аргумента s тип — string

Код не компилируется и выдает ошибку компиляции на данной строке вследствие несовпадения типов

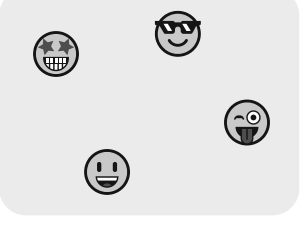
Воспользовавшись системой типов, мы из проблемы времени выполнения, которая могла проявиться при промышленной эксплуатации (и повлиять на наших клиентов), сделали безобидную проблему этапа компиляции, которую просто нужно исправить перед развертыванием кода. Модуль проверки типа гарантирует, что яблоки не будут передаваться в качестве апельсинов; а значит, растет ошибкоустойчивость кода.

Ошибки возникают, когда программа переходит в *некорректное состояние*, то есть текущее сочетание всех ее действующих переменных некорректно по какой-либо причине. Один из методов, позволяющих избавиться от части таких некорректных состояний, — уменьшение пространства состояний за счет ограничения количества возможных значений переменных, как показано на рис. 1.4.



Тип, допускающий больше минимально необходимого количества значений

x = 💀 ; // Плохо



Тип, ограниченный только корректными значениями

x = 💀 ; // Ошибка компиляции

Рис. 1.4. Благодаря правильному объявлению типа можно запретить некорректные значения. Первый тип слишком широк и допускает нежелательные нам значения. Второй тип — более жестко ограниченный — не скомпилируется, если код попытается присвоить переменной нежелательное значение

Пространство состояний (state space) работающей программы можно описать как сочетание всех вероятных значений всех ее действующих переменных. То есть декартово произведение типов всех переменных. Напомню, что тип переменной можно рассматривать как множество ее возможных значений. Декартово произведение двух множеств представляет собой множество, состоящее из всех их упорядоченных пар элементов.

БЕЗОПАСНОСТЬ

Важный побочный результат запрета на потенциальные некорректные состояния — повышение безопасности кода. В основе множества атак лежит выполнение передаваемых пользователем данных, переполнение буфера и другие подобные методики, опасность которых нередко можно уменьшить за счет достаточно сильной системы типов и хороших определений типов.

Корректность кода не исчерпывается исправлением невинных ошибок в коде с целью предотвратить атаки злоумышленников.

1.3.2. Неизменяемость

Неизменяемость (immutability) — еще одно свойство, тесно связанное с представлением о нашей работающей системе как о движении по пространству состояний. Вероятность ошибок можно снизить, если при нахождении системы в заведомо хорошем состоянии не допускать его изменений.

Рассмотрим простой пример, в котором попытаемся предотвратить деление на ноль с помощью проверки значения делителя и генерации ошибки в случае, когда оно равно `0`, как показано в листинге 1.4. Если же значение может меняться после нашей проверки, то она теряет всякий смысл.

Листинг 1.4. «Плохое» изменение значения

```
function safeDivide(): number {
    let x: number = 42;

    if (x == 0) throw new Error("x should not be 0");   ← Проверяем допустимость x

    x = x - 42;   ← Ошибка в программе:
                    после проверки x становится равен 0

    return 42 / x;   ← Деление на 0 приводит
                       к значению Infinity[1]
}
```

В настоящих программах подобное случается регулярно, причем часто довольно неожиданным образом: переменная меняется, скажем, конкурентным потоком выполнения или другой вызванной функцией. Как и в этом примере, сразу после изменения значения все гарантии, которые мы надеялись получить от наших проверок,

[1] Стандартный встроенный объект JavaScript (и TypeScript), олицетворяет бесконечное значение. — *Примеч. пер.*

теряются. Если же сделать `x` константой, как в листинге 1.5, то компилятор вернет ошибку при попытке изменить ее значение.

Листинг 1.5. Неизменяемость

```
function safeDivide(): number {
    const x: number = 42;

    if (x == 0) throw new Error("x should not be 0");

    x = x - 42;

    return 42 / x;
}
```

x объявляется с указанием ключевого слова const вместо let

Эта строка больше не компилируется, поскольку x — неизменяемая и повторное присвоение ей значения недопустимо

Теперь компилятор отвергает некорректный код, выводя следующее сообщение об ошибке: `Cannot assign to 'x' because it is a constant` (Присвоение значения переменной `x` невозможно, поскольку она является константой).

В смысле представления в оперативной памяти разницы между изменяемой и неизменяемой `x` нет. Свойство константности значит что-то только для компилятора. Это свойство, обеспечиваемое системой типов.

Указание на неизменяемость состояния с помощью добавления ключевого слова `const` в описание типа предотвращает те изменения значений, при которых теряются гарантии, полученные благодаря предыдущим проверкам. Особенно полезна неизменяемость в случае конкурентного выполнения, поскольку делает невозможной состояние гонки.

Оптимизация компиляторов обеспечивает выдачу более эффективного кода в случае неизменяемых переменных, так как их значения можно встраивать в код. В некоторых функциональных языках программирования все данные — неизменяемые: функции принимают на входе какие-либо данные и возвращают другие, никогда не меняя входных. При этом достаточно один раз проверить значение переменной и убедиться в ее хорошем состоянии с целью гарантировать, что она будет находиться в хорошем состоянии на протяжении всего жизненного цикла. Конечно, при этом приходится идти на (не всегда желательный) компромисс: копировать данные, с которыми в противном случае можно было бы работать, не прибегая к дополнительным структурам данных.

Впрочем, не всегда имеет смысл делать все данные неизменяемыми. Тем не менее неизменяемость как можно большего числа данных может резко снизить вероятность возникновения таких проблем, как несоответствие заранее заданным условиям и состояние гонки по данным.

1.3.3. Инкапсуляция

Инкапсуляция (encapsulation) — сокрытие части внутреннего устройства кода в функции, классе или модуле. Как вы, вероятно, знаете, инкапсуляция — желательное свойство, она помогает понижать сложность: код разбивается на меньшие компоненты, каждый из которых предоставляет доступ только к тому, что действительно нужно, а подробности реализации скрываются и изолируются.

В листинге 1.6 мы расширим пример безопасного деления, превратив его в класс, который старается гарантировать отсутствие деления на `0`.

Листинг 1.6. Недостаточная инкапсуляция

```
class SafeDivisor {
    divisor: number = 1;

    setDivisor(value: number) {
        if (value == 0) throw new Error("Value should not be 0");

        this.divisor = value;
    }

    divide(x: number): number {
        return x / this.divisor;
    }
}

function exploit(): number {
    let sd = new SafeDivisor();

    sd.divisor = 0;
    return sd.divide(42);
}
```

Проверяем значение перед присваиванием, чтобы гарантировать ненулевой делитель

Деления на 0 не должно быть

Поскольку член класса divisor — публичный, проверку можно обойти

В результате деления на 0 возвращается Infinity

В данном случае мы не можем сделать делитель неизменяемым, поскольку хотим, чтобы у вызывающего наш API кода была возможность его обновлять. Проблема такова: вызывающая сторона может обойти проверку на `0` и непосредственно задать любое значение для `divisor`, так как он для них доступен. Эту проблему в данном случае можно решить, объявив его в качестве `private` и ограничив его область видимости классом, как показано в листинге 1.7.

Листинг 1.7. Инкапсуляция

```
class SafeDivisor {
    private divisor: number = 1;

    setDivisor(value: number) {
        if (value == 0) throw new Error("Value should not be 0");
        this.divisor = value;
    }

    divide(x: number): number {
        return x / this.divisor;
    }
}

function exploit() {
    let sd = new SafeDivisor();

    sd.divisor = 0;
    sd.divide(42);
}
```

Теперь этот член класса стал приватным

Данная строка не скомпилируется, поскольку на divisor больше нельзя ссылаться вне класса

Представление в оперативной памяти приватных и публичных членов класса одинаково; проблемный код не компилируется во втором примере просто благодаря указанию типа. На самом деле `public`, `private` и другие модификаторы видимости — свойства соответствующего типа.

Инкапсуляция (сокрытие информации) позволяет разбивать логику программы и данные на публичный интерфейс и непубличную реализацию. Это очень удобно в больших системах, поскольку при работе с интерфейсами (абстракциями) требуется меньше умственных усилий, чтобы понять конкретный фрагмент кода. Желательно анализировать и понимать код на уровне интерфейсов компонентов, а не всех их нюансов реализации. Полезно также ограничивать область видимости непубличной информации, чтобы внешний код не мог их модифицировать попросту вследствие отсутствия доступа.

Инкапсуляция существует на множестве уровней: сервис предоставляет доступ к своему API в виде интерфейса, модуль экспортирует свой интерфейс и скрывает нюансы реализации, класс делает видимыми только публичные члены класса и т. д. Чем слабее связь между двумя частями кода, тем меньший объем информации они разделяют. Благодаря этому усиливаются гарантии компонента относительно его внутренних данных, поскольку никакой внешний код не может их модифицировать, не прибегая к использованию интерфейса компонента.

1.3.4. Компонуемость

Допустим, нам требуется найти первое отрицательное число в числовом массиве и первую строку из одного символа в символьном массиве. Не прибегая к разбиению этой задачи на две части и последующему их объединению в единую систему, мы получили бы в итоге две функции: `findFirstNegativeNumber()` и `findFirstOneCharacterString()`, показанные в листинге 1.8.

Листинг 1.8. Некомпонуемая система

```
function findFirstNegativeNumber(numbers: number[])
    : number | undefined {
    for (let i of numbers) {
        if (i < 0) return i;
    }
}

function findFirstOneCharacterString(strings: string[])
    : string | undefined {
    for (let str of strings) {
        if (str.length == 1) return str;
    }
}
```

Эти две функции ищут первое отрицательное число и первую строку из одного символа соответственно. Если подобных элементов не найдено, то функции возвращают `undefined` (неявно, путем выхода из функции без оператора `return`).

Если появится новое требование к системе, например, заносить в журнал ошибку в случае невозможности найти искомый элемент, то придется обновить описание обеих функций, как показано в листинге 1.9.

Листинг 1.9. Обновление некомпонуемой системы

```
function findFirstNegativeNumber(numbers: number[])
    : number | undefined {
    for (let i of numbers) {
        if (i < 0) return i;
    }
    console.error("No matching value found");
}

function findFirstOneCharacterString(strings: string[])
    : string | undefined {
    for (let str of strings) {
        if (str.length == 1) return str;
    }
    console.error("No matching value found");
}
```

Данный вариант явно не оптимальный. Что, если мы забудем обновить код в одном из мест? Подобные проблемы усугубляются в больших системах. Судя по виду этих функций, алгоритм в них один и тот же; но в одном случае мы работаем с числами с одним условием, а в другом — со строками с другим условием. Можно написать обобщенный алгоритм с параметризацией по типу обрабатываемых данных и проверяемому условию, как показано в листинге 1.10. Подобный алгоритм не зависит от других частей системы, и его можно анализировать отдельно.

Листинг 1.10. Компонуемая система

```
function first<T>(range: T[], p: (elem: T) => boolean)
    : T | undefined {
    for (let elem of range) {
        if (p (elem)) return elem;
    }
}

function findFirstNegativeNumber(numbers: number[])
    : number | undefined {
    return first(numbers, n => n < 0);
}

function findFirstOneCharacterString(strings: string[])
    : string | undefined {
    return first(strings, str => str.length == 1);
}
```

Не волнуйтесь, если синтаксис немного непривычен; мы обсудим встраиваемые функции (такие как `n => n < 0`) в главе 5 и обобщенные функции — в главах 9 и 10.

Для добавления в эту реализацию журналирования достаточно обновить реализацию функции `first`. Причем если мы придумаем более эффективную реализацию алгоритма, то нужно будет лишь обновить реализацию, и этим автоматически воспользуются все вызывающие функции.

Как мы увидим в главе 10, когда будем обсуждать обобщенные алгоритмы и итераторы, эту функцию можно обобщить еще больше. Пока что она работает с массивом элементов типа `T`, но ее можно обобщить на обход произвольной структуры данных.

Если код некомпонуемый, то понадобится отдельная функция для каждого типа данных, структуры данных и условия, хотя все они, по сути, реализуют одну абстракцию. Возможность абстрагирования с последующим сочетанием и комбинированием компонентов существенно снижает дублирование. Выражать подобные абстракции позволяют обобщенные типы данных.

Возможность сочетания независимых компонентов превращает систему в модульную и уменьшает количество требующего сопровождения кода. Значение компонуемости растет по мере роста объема кода и числа компонентов. Части компонуемой системы сцеплены слабо; в то же время код в отдельных подсистемах не дублируется. Для учета новых требований обычно достаточно обновить один компонент вместо того, чтобы проводить масштабные изменения по всей системе. В то же время для понимания подобной системы требуется меньше мыслительных затрат, поскольку ее части можно анализировать по отдельности.

1.3.5. Читабельность

Код читают намного большее количество раз, чем пишут. Благодаря типизации становится понятно, какие аргументы ожидает функция, каковы предварительные условия для обобщенного алгоритма, какой интерфейс реализует класс и т. д. Ценность этой информации заключается в возможности провести анализ кода по отдельным частям: по одному виду определения, не обращаясь к исходному коду вызывающих и вызываемых функций, можно легко понять, как должен работать код.

Важную роль в этом процессе играют наименования и комментарии, но типизация добавляет в него дополнительный слой информации, позволяя именовать ограничения. Взглянем на нетипизированное объявление функции `find()` в листинге 1.11.

Листинг 1.11. Нетипизированная функция find()

```
declare function find(range: any, pred: any): any;
```

Из описания этой функции непросто понять, какие аргументы она ожидает на входе. Необходимо читать реализацию, пробовать различные параметры и смотреть, не получим ли мы на выходе ошибку во время выполнения, либо надеяться, что все описано в документации.

Сравните с предыдущим объявлением следующий код (листинг 1.12).

Листинг 1.12. Типизированная функция find()

```
declare function first<T>(range: T[],
    p: (elem: T) => boolean): T | undefined;
```

Из этого описания сразу понятно, что для произвольного типа `T` необходимо передать в качестве аргумента `range` массив `T[]` и функцию, принимающую `T` и возвращающую `boolean` в качестве аргумента `p`. Кроме того, сразу же понятно, что функция возвращает `T` или `undefined`.

Вместо поиска реализации или чтения документации достаточно прочесть это объявление функции, чтобы понять, какие именно типы аргументов нужно передавать. Это существенно снижает нашу когнитивную нагрузку благодаря тому, что функция рассматривается как самодостаточная отдельная сущность. Задание подобной информации о типе явным образом, видимым не только компилятору, но и разработчику, намного облегчает понимание кода.

В большинстве современных языков программирования существуют какие-либо *правила вывода типов* (type inference), то есть определения типа переменной по контексту. Это удобно, поскольку позволяет снизить объем требуемого кода, но может превращаться в проблему, если код становится легко понятным для компилятора, но слишком запутанным для людей. Явно прописанный тип намного ценнее комментария, так как его соблюдение обеспечивает компилятор.

1.4. Разновидности систем типов

В настоящее время в большинстве языков программирования и сред выполнения есть типизация в той или иной форме. Мы уже давно осознали, что возможность интерпретировать код как данные и данные как код может привести к катастрофическим последствиям. Основное различие между современными системами типов состоит в том, когда проверяются типы данных, и в степени строгости этих проверок.

При статической типизации проверка совершается во время компиляции, так что по завершении последней гарантируются правильные типы значений во время выполнения. Напротив, при динамической типизации проверка типов данных откладывается до выполнения, поэтому несовпадения типов становятся ошибками времени выполнения.

При сильной типизации производится очень мало преобразований типов (а то и вообще не производится), а менее сильные системы типов допускают больше неявных преобразований типов данных.

1.4.1. Динамическая и статическая типизация

JavaScript — язык с динамической типизацией, а TypeScript — со статической. На самом деле TypeScript был создан именно для добавления статической проверки типов в JavaScript. Превращение ошибок времени выполнения в ошибки компиляции, особенно в больших приложениях, улучшает сопровождаемость и отказоустойчивость кода. Эта книга посвящена статической типизации и статическим языкам программирования, но полезно разобраться и в динамической модели.

Динамическая типизация не предполагает никаких ограничений типов во время компиляции. Обиходное название *«утиная типизация»* (duck typing) возникло из

фразы «Если нечто ходит как утка и крякает как утка то, значит, это утка». Переменная может свободно применяться в коде как угодно, типизация происходит на этапе выполнения. Динамическую типизацию можно имитировать в TypeScript с помощью ключевого слова `any`, которое позволяет использовать нетипизированные переменные.

Реализуем функцию `quacker()`, принимающую на входе аргумент `duck` типа `any` и вызывающую для него функцию `quack()`. Все прекрасно работает, если у передаваемого объекта есть метод `quack()`. Если же передать нечто «не умеющее крякать» (без метода `quack()`), то получим `TypeError` времени выполнения, как показано в листинге 1.13.

Листинг 1.13. Динамическая типизация

```
function quacker(duck: any) {
    duck.quack();
}

quacker({quack: function () {console.log("quack"); }});
quacker(42);
```

Функция принимает аргумент типа any и потому обходит проверку типа на этапе компиляции

Мы передаем объект, содержащий метод quack(), так что в результате вызова в консоль выводится quack

Этот вызов приводит к ошибке во время выполнения: TypeError: duck.quack is not a function (Ошибка типа: duck.quack не является функцией)

При статической типизации, с другой стороны, проверка типов производится на этапе компиляции, так что попытка передать аргумент не того типа вызывает ошибку компиляции. Для полноценного использования возможностей статической типизации TypeScript можно усовершенствовать код, объявив в нем интерфейс `Duck` и указав соответствующий тип аргумента функции, как показано в листинге 1.14. Обратите внимание: в TypeScript не обязательно явным образом указывать, что мы реализуем интерфейс `Duck`, лишь бы был метод `quack()`. Если функция `quack()` есть, то компилятор считает интерфейс реализованным. В других языках программирования пришлось бы явным образом объявить, что класс реализует этот интерфейс.

Листинг 1.14. Статическая типизация

```
interface Duck {
    quack(): void;
}

function quacker(duck: Duck) {
    duck.quack();
}

quacker({quack: function () {console.log("quack"); }});
quacker(42);
```

Объявление интерфейса для объекта, у которого должен быть метод quack()

У модифицированной функции теперь должен быть аргумент типа Duck

Ошибка компиляции: Argument of type '42' is not assignable to parameter of type 'Duck' (Невозможно присвоить параметру типа 'Duck' аргумент типа '42')

Основное преимущество статической типизации — перехват подобных ошибок на этапе компиляции до того, как они вызовут сбой работающей программы.

1.4.2. Слабая и сильная типизации

При описания систем типов часто можно встретить термины *«сильная типизация»*[1] (strong typing) и *«слабая типизация»* (weak typing). Сила системы типов определяется степенью строгости соблюдения ею ограничений типов. Слабая система неявно преобразует значения из их фактических типов в типы, ожидаемые там, где они используются.

Задумайтесь: «молоко» равно «белое»? В сильно типизированном мире ответ на этот вопрос: нет, молоко — жидкость и сравнивать ее с цветом бессмысленно. В слабо типизированном мире можно сказать: «Ну, цвет молока — белый, так что да, молоко равно белому». В сильно типизированном мире можно явным образом преобразовать молоко в цвет, задав вопрос вот так: «Равен ли цвет молока белому?» В слабо типизированном мире это уточнение не требуется.

JavaScript — слабо типизированный язык. Чтобы это увидеть, достаточно воспользоваться типом `any` в TypeScript и делегировать типизацию во время выполнения JavaScript. В JavaScript есть два оператора проверки на равенство: `==`, проверяющий равенство двух значений, и `===`, проверяющий равенство как значений, так и их типов (листинг 1.15). Поскольку JavaScript — слабо типизированный язык, выражение вида `"42" == 42` равно `true`. Это довольно странно, ведь `"42"` — текстовое значение, а `42` — число.

Листинг 1.15. Слабая типизация

```
const a: any = "hello world";
const b: any = 42;

console.log(a == b);

console.log("42" == b);

console.log("42" === b);
```

Выводит false, хотя сравнение строки с числом допустимо

Выводит true; среда выполнения JavaScript неявно преобразует значения к одному типу

Выводит false; оператор === сравнивает и типы тоже

Неявные преобразования типов удобны тем, что не нужно писать много лишнего кода для явного преобразования из одного типа в другой, но и опасны, поскольку во многих случаях такие трансформации нежелательны и неожиданны для программиста. Благодаря сильной типизации TypeScript не скомпилирует ни одну из предыдущих операций сравнения, если объявить должным образом переменную `a` с типом `string` и переменную `b` с типом `number`, как показано в листинге 1.16.

Все эти операции сравнения вернут ошибку `"This condition will always return 'false' since the types 'string' and 'number' have no overlap"`. (Это условие всегда возвращает `false`, поскольку типы `'string'` и `'number'` не пересекаются.) Модуль проверки типа обнаруживает, что мы пытаемся сравнить значения различных типов, и забраковывает код.

---

[1] В русскоязычной литературе часто также называется строгой типизацией. — *Примеч. пер.*

Листинг 1.16. Сильная типизация

```
const a: string =c"hello world";
const b: number = 42;

console.log(a == b);

console.log("42" == b);

console.log("42" === b);
```

Переменные a и b больше не объявлены с типом any, так что должны пройти проверку типов

Ни одна из трех операций сравнения не скомпилируется, поскольку TypeScript не разрешает сравнения различных типов

Работать со слабой системой типов проще в краткосрочной перспективе, ведь эта система не заставляет программистов явно преобразовывать типы значений, однако она не дает тех гарантий, которые предоставляет сильная система. Большинство описанных в этой главе преимуществ и используемые в оставшейся части данной книги методики потеряют свою эффективность, если не подкрепить их должным образом.

Обратите внимание: хотя система типов может быть либо динамической (проверка типов во время выполнения), либо статической (проверка типов во время компиляции), существует целый диапазон степеней ее строгости: чем менее явные преобразования она производит, тем слабее система. В большинстве систем типов, даже сильных, есть какие-либо ограниченные возможности неявного приведения типов для считающихся безопасными преобразований. Распространенный пример — преобразование к `boolean`: `if (a)` скомпилируется, даже если `a` — `number` или относится к ссылочному типу. Еще один пример — *расширяющее приведение типов* (widening cast), о котором мы поговорим подробнее в главе 4. Для числовых значений в TypeScript используется только тип `number`, но в других языках, когда, допустим, передается восьмибитное значение при необходимом 16-битном целом числе, преобразование обычно выполняется автоматически, так как риска порчи данных нет (16-битное целое число может содержать любое значение, содержащееся в восьмибитном числе, и не только его).

1.4.3. Вывод типов

В некоторых случаях компилятор может вывести, исходя из контекста, тип переменной или функции, не указанный явным образом. Если присвоить переменной значение `42`, например, то компилятор TypeScript может вывести, что ее тип — `number`, и нам не придется указывать тип. Это позволит увеличить прозрачность и понятность читателям кода, но соответствующая нотация необязательна.

Аналогично, если функция возвращает значения одного типа во всех операторах `return`, то указывать возвращаемый тип явно в описании функции не нужно. Компилятор может вывести эту информацию из кода, как показано в листинге 1.17.

В отличие от динамической типизации, которая производится только на этапе выполнения, в подобных случаях типизация определяется и проверяется на этапе компиляции, хотя явным образом описывать типы не нужно. При неоднозначности типизации компилятор выдаст ошибку и попросит нас указать нотацию типов более явным образом.

Листинг 1.17. Вывод типа

```
function add(x: number, y: number) {
    return x + y;
}

let sum = add(40, 2);
```

У этой функции не указан явный возвращаемый тип, но компилятор определяет, что данный тип — number

Тип переменной sum не объявлен явным образом, а выводится компилятором

1.5. В этой книге

Сильная статическая система типов позволяет писать более корректный, лучше компонуемый и читабельный код. В данной книге мы рассмотрим основные возможности подобных современных систем типов с упором на их практическое применение.

Мы начнем с *простых типов данных* (primitive types), готовых для применения типов, доступных в большинстве языков программирования. Обсудим, как правильно их использовать и избежать распространенных ловушек. В ряде случаев будут показаны способы реализации некоторых из этих типов данных при отсутствии их нативной реализации в языке программирования.

Далее мы обсудим компонуемость и возможность сочетания простых типов данных в целях создания целой вселенной типов, необходимых для предметной области конкретной задачи. Существует множество способов сочетания типов данных, и вы узнаете, как выбрать правильный инструмент в зависимости от конкретной решаемой задачи.

Затем будет рассказано о *функциональных типах данных* (function types) и новых реализациях, обязанных своим появлением возможностям типизации функций и использования их аналогично обычным значениям. Функциональное программирование — весьма обширная тема, так что я не стану пытаться изложить ее во всей полноте, мы позаимствуем из нее некоторые полезные понятия и применим их к нефункциональному языку программирования для решения реальных задач.

Следующий этап эволюции систем типов после типизации значений, компоновки типов и типизации функций — *создание подтипов* (subtyping). Мы обсудим, какие качества делают тип подтипом другого типа, и попытаемся применить в нашем коде некоторые концепции объектно-ориентированного программирования. Обсудим наследование, компоновку и такой менее традиционный инструмент, как примеси.

Далее будет рассказано про *обобщенные типы данных* (generics), благодаря которым возможны переменные типов и параметризация кода типом данных. Обобщенные типы представляют собой совершенно новый уровень абстракции и компонуемости, расцепляя данные с их структурами, а структуры — с алгоритмами и делая вероятными адаптивные алгоритмы.

И наконец, обсудим *типы более высокого рода* (higher kinded types) — следующий уровень абстракции, параметризацию обобщенных типов данных. Типы более высокого рода представляют собой формализацию таких структур данных, как моноиды

и монады. В настоящее время многие языки программирования не поддерживают типы более высокого рода, но их широкое применение в таких языках, как Haskell, и растущая популярность в конце концов должны привести и к внедрению их в более традиционные языки программирования.

Резюме

- *Тип* — классификация данных по возможным операциям над ними, их смыслу и набору допустимых значений.
- *Система типов* — набор правил назначения типов элементам языка программирования.
- Тип ограничивает диапазон принимаемых переменной значений, так что в некоторых случаях ошибка времени выполнения превращается в ошибку компиляции.
- *Неизменяемость* — свойство данных, возможное благодаря типизации и гарантирующее, что переменная не поменяется, когда не должна.
- *Видимость* — еще одно свойство уровня типа, определяющее, к каким данным есть доступ у тех или иных компонентов.
- Обобщенное программирование предоставляет широкие возможности расцепления и повторного использования кода.
- Указание нотаций типов упрощает понимание кода.
- Динамическая («утиная») типизация — определение типа на этапе выполнения.
- При статической типизации типы проверяются во время компиляции и перехватываются ошибки, которые в противном случае могли бы возникнуть во время выполнения.
- Строгость системы типов определяется числом допустимых неявных преобразований типов.
- Современные модули проверки типов включают обладающие широкими возможностями алгоритмы вывода, которые позволяют определять типы переменных, функций и т. д. без явного их указания в коде.

В главе 2 мы рассмотрим простые типы данных — простейшие стандартные блоки систем типов. Научимся избегать некоторых распространенных ошибок, возникающих при использовании этих типов, а также узнаем, как создать практически любую структуру данных из массивов и ссылок.

2 Базовые типы данных

В этой главе

- Основные простые типы данных и их использование.
- Вычисление булевых значений.
- Ловушки числовых типов и кодирования текста.
- Базовые типы для создания структур данных.

В качестве внутреннего представления данных в компьютере используются последовательности битов. Смысл этим последовательностям придают типы. В то же время типы служат для ограничения диапазонов допустимых значений элементов данных. Системы типов содержат наборы простых (встроенных) типов данных и наборы правил их сочетания.

В этой главе мы рассмотрим часто встречающиеся простые типы данных (пустой, единичный, булев тип, числа, строки, массивы и ссылки), способы их применения и распространенные ловушки. Хотя мы используем простые типы данных ежедневно, существуют малозаметные нюансы, которые необходимо учитывать для эффективного применения этих типов. Например, существует возможность сокращенного вычисления булевых выражений, а при вычислении числовых выражений может происходить переполнение.

Мы начнем с простейших типов, практически не несущих информации, и постепенно перейдем к типам, представляющим данные с помощью различных видов

кодирования. Наконец, рассмотрим массивы и ссылки — стандартные блоки всех прочих более сложных структур данных.

2.1. Проектирование функций, не возвращающих значений

Если рассматривать типы как множества вероятных значений, то возникает вопрос: а существует ли тип, соответствующий пустому множеству? Оно не содержит элементов, так что невозможно будет создать экземпляр этого типа. Будет ли польза от такого типа?

2.1.1. Пустой тип

Посмотрим, сможем ли мы описать как часть библиотеки утилит функцию, которая, получив сообщение в качестве параметра, заносила бы в журнал факт возникновения ошибки, включая метку даты/времени и сообщение, после чего генерировала бы исключение, как показано в листинге 2.1. Такая функция является просто оберткой для `throw`, поэтому не должна возвращать управление.

Листинг 2.1. Генерация и журналирование ошибки в случае отсутствия файла конфигурации

```
const fs = require("fs");

function raise(message: string): never {
    console.error(`Error "${message}" raised at ${new Date()}`);
    throw new Error(message);
}

function readConfig(configFile: string): string {
    if (!fs.existsSync(configFile))
        raise(`Configuration file ${configFile}missing`);

    return fs.readFileSync(configFile, "utf-8");
}
```

Функция никогда не возвращает управление (всегда генерирует исключение), так что ее возвращаемый тип — never

Пример использования: если файл конфигурации не найден, то функция должна занести информацию об этом в журнал и сгенерировать ошибку

Обратите внимание: возвращаемый тип функции в данном примере — `never`. Благодаря этому читателям кода понятно, что функция `raise` никогда не должна возвращать значение. Более того, если кто-нибудь потом случайно изменит описание функции, добавив оператор `return`, то код перестанет компилироваться. Типу `never` нельзя присвоить абсолютно никакое значение, поэтому задуманное поведение функции обеспечивает компилятор и гарантирует, что она не будет возвращать управление.

Подобный тип данных называется *«необитаемым»* (uninhabitable type), или *пустым типом данных* (empty type), поскольку создать его экземпляр невозможно.

ПУСТОЙ ТИП ДАННЫХ

Пустой тип — это тип данных, у которого не может быть никакого значения: множество его вероятных значений — пустое. Задать значение переменной такого типа невозможно. Пустой тип уместен как символ невозможности чего-либо, например, в качестве возвращаемого типа функции, которая никогда не возвращает значения (генерирует исключение или содержит бесконечный цикл).

«Необитаемый» тип данных используется для объявления функций, которые никогда не возвращают значений. Функция может не возвращать значения по нескольким причинам: генерация исключения по всем ветвям кода, работа в бесконечном цикле или возникновение фатального сбоя программы. Все эти сценарии допустимы. Например, может понадобиться реализовать функцию, производящую журналирование или отправляющую телеметрические данные перед генерацией исключения либо аварийным выходом из программы в случае неустранимой ошибки. Или может возникнуть необходимость в коде, который бы непрерывно работал в цикле вплоть до момента останова всей системы, например, для обработки событий системы.

Объявление подобной функции как возвращающей `void` (тип, используемый в большинстве языков программирования для указания на отсутствие осмысленного значения) только вводит читателя в заблуждение. Наша функция не просто не возвращает осмысленное значение, она вообще ничего не возвращает!

Незавершающиеся функции

Пустой тип может показаться тривиальным, но демонстрирует фундаментальное различие между математикой и информатикой: в математике нельзя определить функцию, отображающую непустое множество в пустое. Это просто лишено смысла. Функции в математике не «вычисляются», они просто «существуют».

Компьютеры, с другой стороны, вычисляют программы; пошагово выполняют инструкции. Компьютер в процессе вычислений может оказаться в бесконечном цикле, выполнение которого никогда не прекратится. Поэтому в компьютерных программах *могут* описываться осмысленные функции отображения в пустое множество, такие как в предыдущих примерах.

Пустой тип имеет смысл использовать везде, где встречаются не возвращающие ничего функции, либо с целью показать явным образом, что никакого значения нет.

Самодельный пустой тип

Далеко не во всех широко распространенных языках программирования есть готовый пустой тип данных наподобие типа `never` в TypeScript. Но в большинстве языков его можно реализовать самостоятельно. Это осуществимо с помощью описания перечисляемого типа, не содержащего никаких элементов или структуры с одним только приватным конструктором, чтобы его нельзя было вызвать.

В листинге 2.2 показано, как можно реализовать пустой тип в TypeScript в виде класса, не допускающего создания экземпляров. Обратите внимание: TypeScript счи-

тает два типа со схожей структурой совместимыми, так что нам придется добавить фиктивное свойство типа `void`, чтобы в прочем коде не могло оказаться значения, которое неявно бы преобразовалось в `Empty`. В прочих языках, например Java и C#, такого дополнительного свойства не требуется, поскольку в них совместимость типов не определяется на основе их формы. Мы обсудим этот вопрос подробнее в главе 7.

Листинг 2.2. Реализация пустого типа в виде невоплощаемого класса

```
declare const EmptyType: unique symbol;

class Empty {
    [EmptyType]: void;
    private constructor() {}
}

function raise(message: string): Empty {
    console.error(`Error "${message}" raised at ${new Date()}`);
    throw new Error(message);
}
```

Подобным специфическим образом в TypeScript обеспечивается невозможность интерпретировать другие объекты той же формы как объекты этого типа

Приватный конструктор гарантирует, что создать экземпляр данного типа в остальном коде невозможно

Эта функция не отличается от предыдущего примера, но теперь вместо never используется Empty

Данный код компилируется, поскольку компилятор выполняет анализ потока команд и определяет, что оператор `return` не нужен. С другой стороны, добавить этот оператор невозможно, поскольку нельзя создать экземпляр класса `Empty`.

2.1.2. Единичный тип

В предыдущем подразделе мы обсуждали функции, никогда ничего не возвращающие. А как насчет функций, которые производят возврат, но не возвращают ничего полезного? Существует множество подобных функций, вызываемых исключительно ради их побочных эффектов: они *производят* определенные действия, меняя какое-либо внешнее состояние, но не выполняют никаких вычислений, результаты которых могли бы вернуть.

Рассмотрим в качестве примера функцию `console.log()`: она выводит свой аргумент в отладочную консоль, но не возвращает никаких осмысленных значений. С другой стороны, по завершении выполнения она *возвращает* управление вызывающей стороне, так что ее возвращаемым типом не может служить `never`.

Классическая функция `"Hello world!"`, приведенная в листинге 2.3, — еще один хороший пример этого. Ее вызывают для вывода в консоль приветствия (то есть ради побочного эффекта), а не в целях возврата значения, так что мы укажем для нее возвращаемый тип `void`.

Листинг 2.3. Функция «Hello world!»

```
function greet(): void {
    console.log("Hello world!");
}

greet();
```

Данная функция выводит в консоль приветствие и не возвращает ничего полезного

Обычно результат подобных функций просто игнорируют

Возвращаемый тип подобных функций называется *единичным типом* (unit type), то есть типом, у которого может быть только одно значение, и в TypeScript и большинстве других языков он называется `void`. Причина, из-за чего обычно не используются переменные типа `void`, а просто производится возврат из `void` функции без указания реального значения, состоит как раз в том, что значение единичного типа неважно.

ЕДИНИЧНЫЙ ТИП

Единичный тип (unit type) — это тип, число вероятных значений которого равно одному. Нет смысла проверять значение переменной подобного типа — оно может быть только одним. Единичный тип используют, когда возвращаемый функцией результат неважен.

Функции, принимающие какое-либо число аргументов, но не возвращающие никакого осмысленного значения, называются также *действиями* (actions), поскольку обычно выполняют одну или несколько операций, которые меняют состояние, или *потребителями* (consumers), так как получают аргументы, но ничего не возвращают.

Самодельный единичный тип

Типы, подобные `void`, есть в большинстве языков программирования, однако некоторые языки рассматривают этот тип особым образом и не позволяют использовать его полностью аналогично другим типам. В подобных случаях можно создать собственный единичный тип, описав перечисляемый тип из одного элемента или класс-одиночку без состояния. Так как переменная единичного типа может принимать только одно значение, неважно, каким это значение будет; все единичные типы эквивалентны. Преобразование из одного единичного типа в другой тривиально, поскольку нет никаких вариантов: единственное значение одного типа соответствует единственному значению другого.

В листинге 2.4 показано, как можно реализовать единичный тип в TypeScript. Как и для самодельного пустого типа, мы воспользуемся свойством типа `void`, чтобы другие типы с совместимой структурой не преобразовывались неявно в `Unit`. В других языках программирования, например Java и C#, это дополнительное свойство не нужно.

Листинг 2.4. Реализация единичного типа в виде класса-одиночки без состояния

```
declare const UnitType: unique symbol;

class Unit {
    [UnitType]: void;
    static readonly value: Unit = new Unit();
    private constructor() {};
}
```

Уникальное свойство гарантирует, что типы аналогичного вида не будут интерпретироваться как Unit

Статическое свойство только для чтения типа Unit — единственный возможный экземпляр Unit

Приватный конструктор гарантирует, что создать экземпляр данного типа в остальном коде невозможно

```
function greet(): Unit {
    console.log("Hello world!”);
    return Unit.value;
}
```

Эквивалентно возвращающей void функции, всегда возвращает одно и то же значение

2.1.3. Упражнения

1. Какой возвращаемый тип должен быть у функции `set()`, принимающей на входе значение и присваивающей его глобальной переменной?
   А. `never`.
   Б. `undefined`.
   В. `void`.
   Г. `any`.
2. Какой возвращаемый тип должен быть у функции `terminate()`, которая немедленно прерывает выполнение программы?
   А. `never`.
   Б. `undefined`.
   В. `void`.
   Г. `any`.

2.2. Булева логика и сокращенные схемы вычисления

За типами, у которых не может быть возможных значений (пустых типов наподобие `never`), и типами с одним вероятным значением (единичных типов вроде `void`) логично следуют типы с двумя такими значениями. Каноническим примером типа с двумя возможными значениями, доступным в большинстве языков программирования, является *булев* (Boolean) тип.

Булевы значения отражают правдивость. Свое название они получили в честь Джорджа Буля (George Boole), придумавшего то, что сегодня носит название булевой алгебры — алгебры, состоящей из истинного значения (`1`), ложного значения (`0`) и логических операций над ними, например `AND`, `OR` и `NOT`.

В некоторых системах типов есть встроенный булев тип со значениями `true` и `false`. Другие системы используют числовые значения, считая, что `0` означает `false`, а любое другое число — `true` (то есть *все, что не ложь, — истина*). В TypeScript есть встроенный тип `boolean`, который может принимать значения `true` и `false`.

Вне зависимости от того, существует ли в конкретном языке простой булев тип или истинность определяется на основе значений других типов, в большинстве языков программирования используется какая-либо форма булевой семантики для *условного ветвления* (conditional branching). В таких операторах, как `if (условие) { ... }`,

выражение между фигурными скобками выполняется, только если в результате вычисления условия получается истина. Условия применяются и в циклах, чтобы понять, продолжать ли итерации или завершить выполнение цикла: `while (условие) { ... }`. Без условного ветвления писать по-настоящему полезный код было бы невозможно. Представьте, как бы вы реализовали простейший алгоритм, например поиск первого четного числа в списке чисел, без циклов или условных операторов.

2.2.1. Булевы выражения

Во многих языках программирования для распространенных булевых операций используются следующие символы: `&&` для `AND`, `||` для `OR` и `!` для `NOT`. Булевы выражения обычно описываются с помощью таблиц истинности (рис. 2.1).

a	b	a && b	a || b	!a
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Рис. 2.1. Таблицы истинности AND, OR и NOT

2.2.2. Схемы сокращенного вычисления

Представьте, что вы хотите создать шлюз для системы комментирования, показанной в листинге 2.5: шлюз отвергает комментарии, отправленные пользователем менее чем через 10 секунд после предыдущего (спам), и комментарии с пустым содержимым (пользователь случайно нажал кнопку `Comment` (Отправить комментарий) до того, как написал что-либо).

Функция-шлюз принимает в качестве аргументов сам комментарий и идентификатор пользователя. Функция `secondsSinceLastComment()` у вас уже реализована; она выполняет запрос к базе данных по заданному идентификатору пользователя и возвращает количество секунд, прошедших с отправки им последнего комментария.

Если оба условия выполнены, то комментарий отправляется в базу данных, если нет — возвращается `false`.

Листинг 2.5 — одна из возможных реализаций подобного шлюза. Обратите внимание на выражение `OR`, в котором возвращается `false`, если предыдущий комментарий был отправлен менее чем 10 секунд назад *или* текущий комментарий пуст.

Другой способ реализации той же логики — поменять два операнда местами, как показано в листинге 2.6. Сначала проверяем, не пуст ли текущий комментарий; а затем проверяем, когда был отправлен предыдущий комментарий, как и в листинге 2.5.

Есть ли преимущество у какой-либо из этих версий? В них описаны одни и те же проверки — только в другом порядке. Оказывается, различие есть. В зависимости от

входных данных версии могут вести себя по-разному во время выполнения вследствие того, как вычисляются булевы выражения.

Листинг 2.5. Шлюз

```
declare function secondsSinceLastComment(userId: string): number;
declare function postComment(comment: string, userId: string): void;

function commentGatekeeper(comment: string, userId: string): boolean {
    if ((secondsSinceLastComment(userId) < 10) || (comment == ""))
        return false;

    postComment(comment, userId);

    return true;
}
```

Функция secondsSinceLastComment() запрашивает в базе данных информацию о том, насколько давно был отправлен предыдущий комментарий пользователя

Функция postComment() записывает комментарий в базу данных

Если хотя бы одно из условий не выполнено, то возвращается false. В противном случае отправляется комментарий и возвращается true

Листинг 2.6. Другой вариант реализации шлюза

```
declare function secondsSinceLastComment(userId: string): number;
declare function postComment(comment: string, userId: string): void;

function commentGatekeeper(comment: string, userId: string): boolean {
    if ((comment == "") || (secondsSinceLastComment(userId) < 10))
        return false;

    postComment(comment, userId);

    return true;
}
```

Эта версия и предыдущая различаются только порядком условий

Большинство компиляторов и сред выполнения оптимизируют булевы выражения с помощью так называемого *сокращенного вычисления* (short circuit). Выражение вида `a AND b` преобразуется в `if a then b else false`. При этом используется таблица истинности для операции `AND`: если первый операнд ложен, то и все выражение ложно, вне зависимости от значения второго операнда. С другой стороны, если первый операнд истинен, то все выражение истинно только в случае истинности и второго операнда.

Аналогичное преобразование производится для выражения `a OR b`, которое превращается в `if a then true else b`. Из таблицы истинности для операции `OR` видим, что если первый операнд истинен, то и все выражение истинно, вне зависимости от значения второго операнда. В противном же случае, когда первый операнд является ложным, все выражение истинно, если истинен второй операнд.

Причина такого преобразования и появления названия *«сокращенное вычисление»* — тот факт, что если вычисление первого операнда дает достаточно информации

для вычисления всего выражения, то значение второго вообще не вычисляется. Шлюзовая функция должна выполнить две проверки. Первая не требует особых затрат ресурсов и проводится с целью убедиться в том, что полученный комментарий не пуст. Вторая — потенциально весьма дорогостоящая, включает запрос к базе комментариев. В листинге 2.5 сначала выполняется запрос к базе данных. Если последний комментарий был отправлен более 10 секунд назад, то сокращенная схема вычислений вообще не станет анализировать текущий комментарий и просто вернет `false`. В листинге 2.6, если текущий комментарий пуст, запрос к базе данных производиться не будет. Вторая версия потенциально может исключить дорогостоящую проверку за счет вычисления другой, гораздо менее затратной.

Это свойство вычисления булевых выражений очень важно, его необходимо учитывать при сочетании условий: сокращенная схема вычислений позволяет избежать вычисления правого выражения в зависимости от результата вычисления левого выражения, так что условия желательно упорядочивать от наименее затратного по возрастающей.

2.2.3. Упражнение

Что будет выведено в результате выполнения следующего кода?

```
let counter: number = 0;

function condition(value: boolean): boolean {
    counter++;
    return value;
}

if (condition(false) && condition(true)) {
    // ...
}
console.log(counter)
```

А. `0`.

Б. `1`.

В. `2`.

Г. Ничего, будет сгенерировано исключение.

2.3. Распространенные ловушки числовых типов данных

В большинстве языков программирования одним из простых типов данных служат числовые типы. Существует несколько нюансов, которые желательно учитывать при работе с числовыми данными. Возьмем в качестве примера простую функцию для вычисления общей стоимости купленных товаров (листинг 2.7). Если пользователь купил три пачки жевательной резинки по 10 центов каждая, то итоговая сумма должна быть 30 центов. Но результат, полученный при некоторых способах применения числовых типов, может вас удивить.

Листинг 2.7. Функция подсчета общей стоимости купленных товаров

```
type Item = {name: string, price: number };

function getTotal(items: Item[]): number {
    let total: number = 0;

    for (let item of items) {
        total += item.price;
    }
    return total;
}

let total: number = getTotal(
    [{name: "Cherry bubblegum", price: 0.10 },
     {name: "Mint bubblegum", price: 0.10 },
     {name: "Strawberry bubblegum", price: 0.10 }]
);

console.log(total == 0.30);
```

Каждому товару соответствует название и цена (число)

Функция getTotal возвращает итоговую сумму в виде числа

Вычисляем суммарную стоимость трех пачек жевательной резинки по 10 центов каждая

Выводится false, хотя логично предположить, что 0,10 + 0,10 + 0,10 = 0,30

Почему же в результате суммирования 0,10 три раза не получается 0,30? Чтобы понять это, нам придется разобраться в том, как в компьютерах представляются числовые типы данных. Две определяющие характеристики числового типа — его ширина и способ кодирования.

Ширина (width) — это количество битов, используемое для представления значения. Может варьироваться от восьми бит (один байт) или даже одного бита до 64 бит и более. Битовая ширина тесно связана с архитектурой процессора: у 64-битного процессора и регистры 64-битные, благодаря чему операции над 64-битными значениями выполняются чрезвычайно быстро. Существует три способа кодирования числа заданной ширины: *беззнаковый двоичный код* (unsigned binary), *дополнительный код* (two's complement) и *IEEE 754*.

2.3.1. Целочисленные типы данных и переполнение

При беззнаковом двоичном кодировании для представления части значения используются все биты. Например, четырехбитное беззнаковое целое число может представлять любое значение от `0` до `15`. В общем случае с помощью *N*-битного беззнакового целого числа можно представить значения от `0` (все биты равны `0`) до 2^N`-1` (все биты равны `1`). На рис. 2.2 показано несколько возможных значений четырехбитного беззнакового целого числа. Последовательность из *N* двоичных цифр ($b^{N-1}b^{N-2}... b^1b^0$) можно преобразовать в десятичное число по формуле $b^{N-1} \times 2^{N-1} + b^{N-2} \times 2^{N-2} + ... + b^1 \times 2^1 + b^0 \times 2^0$.

Это очень простой способ кодирования, который, впрочем, позволяет представлять только положительные числа. Представление отрицательных чисел возможно с помощью другого способа. Обычно для этой цели используется так называемый дополнительный код. Представление положительных чисел — точно такое же, как показано выше, а отрицательные кодируются путем вычитания их модуля из 2^N, где *N* — количество битов. На рис. 2.3 показано несколько возможных значений четырехбитного знакового целого числа.

Значение	Четырехбитное беззнаковое кодирование
0	0000
1	0001
2	0010
10	1010
15	1111

Минимально возможное значение; все биты равны 0

Максимально возможное значение; все биты равны 1

Рис. 2.2. Четырехбитное беззнаковое кодирование целых чисел. Минимально возможное значение при равенстве всех четырех бит 0 равно 0. Максимальное значение при равенстве всех четырех бит 1 равно 15 ($1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$)

Значение	Четырехбитное беззнаковое кодирование
–8	1000
–3	1101
0	0000
3	0011
7	0111

Минимально возможное значение; все биты равны 0, за исключением бита знака

Максимально возможное значение; все биты равны 1, за исключением бита знака

Рис. 2.3. Четырехбитное знаковое кодирование целых чисел. Значение –8 кодируется как $2^4 - 8$ (1000 в двоичной системе счисления), а –3 — как $2^4 - 3$ (1101 в двоичной системе счисления). Первый бит всегда равен 1 для отрицательных чисел и 0 для положительных

При таком способе кодирования у всех отрицательных чисел первый бит будет равен `1`, а у всех положительных — `0`. С помощью четырехбитного знакового целого можно отражать значения от `-8` до `7`. Чем больше битов используется для представления значения, тем большее значение можно представить.

Переполнение и потеря значимости

А что происходит, если результат арифметической операции не помещается в заданное число битов? Что, если мы пытаемся с помощью четырехбитного беззнакового кодирования сложить 10 с 10, хотя максимальное значение, которое можно представить с помощью четырех бит, — `15`?

Подобная ситуация называется *арифметическим переполнением* (arithmetic overflow). Противоположная ситуация, когда число оказывается слишком маленьким для того, чтобы его можно было представить, называется *потерей значимости* или *исчезновением порядка* (arithmetic underflow). В различных языках программирования эти ситуации решаются по-разному (рис. 2.4).

Основные три способа решения проблем арифметического переполнения и потери значимости — возврат на ноль (wrap around), останов на максимальном значении (saturation) и вывод сообщения об ошибке (error out).

Рис. 2.4. Различные способы решения проблемы арифметического переполнения. Одометр переходит с 999 999 обратно на 0; диск телефона останавливается на максимально возможном значении; карманный калькулятор выводит на экран сообщение об ошибке (Error) и прекращает работу

Аппаратное обеспечение обычно производит *возврат на ноль*, то есть просто отбрасывает лишние биты. В случае четырехбитного беззнакового целого числа, если биты выглядят как `1111` и мы пытаемся прибавить 1, результат будет равен `10000`, но, поскольку есть только четыре бита, один отбрасывается и остается `0000`, то есть просто `0`. Это самый эффективный способ борьбы с переполнением, но и самый опасный, так как может привести к неожиданным результатам. Это все равно что прибавить один доллар к имеющимся 15 и остаться с нулем.

Второй способ — *останов на максимальном значении*. Если результат превышает максимально представимое значение, то мы просто останавливаемся на данном максимуме. Это хорошо согласуется с реальным миром: при реле температуры, рассчитанном на определенную максимальную температуру, попытка сделать теплее ничего не даст. С другой стороны, при использовании этого метода арифметические операции перестают быть ассоциативными. Если наше максимальное значение равно `7`, то `7 + (2 – 2) = 7 + 0 = 7`, но `(7 + 2) – 2 = 7 – 2 = 5`.

Третья возможность — *выдача сообщения об ошибке* в случае переполнения. Это наиболее безопасный подход, впрочем имеющий недостаток: необходимо проверять все до единой арифметические операции и обрабатывать исключения при любых арифметических действиях.

Обнаружение переполнения и потери значимости

В зависимости от используемого языка программирования можно обрабатывать арифметическое переполнение и потерю значимости любым из описанных способов. Если же для вашего сценария требуется другой способ, не тот, что принят в языке по умолчанию, то придется проверять возможное переполнение/потерю значимости в операции и обрабатывать данный сценарий отдельно. Фокус в том, чтобы не выйти при этом из диапазона допустимых значений.

Например, чтобы проверить, не приведет ли сложение значений `a` и `b` к переполнению/потере значимости диапазона `[MIN, MAX]`, необходимо убедиться, что не получится `a + b < MIN` (при сложении двух отрицательных чисел) или `a + b > MAX`.

Если значение `b` больше нуля, то ситуация `a + b < MIN` невозможна в принципе, так как мы увеличиваем значение `a`, а не уменьшаем. В этом случае необходимо

проверять только возможность переполнения. Вычитая с обеих сторон неравенства `b`, можно переписать `a + b > MAX` в виде `a > MAX – b`. А поскольку мы вычитаем положительное число, сумма становится меньше, поэтому риска переполнения нет (`MAX – b` заведомо находится в диапазоне `[MIN, MAX]`). Так что переполнение происходит, если `b > 0` и `a > MAX – b`.

Если значение `b` меньше нуля, то ситуация `a + b > MAX` невозможна в принципе, так как мы уменьшаем значение `a`, а не увеличиваем. В этом случае достаточно проверить только на потерю значимости. Вычитая с обеих сторон неравенства `b`, можно переписать `a + b < MIN` в виде `a < MIN – b`. А поскольку мы вычитаем отрицательное число, значение становится больше, поэтому риска потери значимости нет (`MIN – b` заведомо находится в диапазоне `[MIN, MAX]`). Так что потеря значимости происходит, если `b < 0` и `a < MIN – b`, как показано в листинге 2.8.

Листинг 2.8. Проверка переполнения при сложении

```
function addError(a: number, b: number,
    min: number, max: number): boolean {
    if (b >= 0) {
        return a > max - b;
    }else {
        return a < min - b;
    }
}
```

Функция принимает в качестве аргументов числа a и b, а также минимальное и максимальное допустимые значения

При b > 0 переполнение происходит, если a > MAX – b

При b < 0 потеря значимости происходит, если a < MIN – b

Аналогичная логика применима при вычитании.

При умножении мы произведем проверку на переполнение и потерю значимости путем деления обеих сторон на `b`. В данном случае необходимо учитывать знаки обоих чисел, поскольку умножение двух отрицательных чисел дает в результате положительное, а умножение отрицательного числа на положительное дает отрицательное.

Переполнение происходит, если:

- `b > 0`, `a > 0` и `a > MAX / b`;
- `b < 0`, `a < 0` и `a < MAX / b`.

Потеря значимости происходит, если:

- `b > 0`, `a < 0` и `a < MIN / b`;
- `b < 0`, `a > 0` и `a > MIN / b`.

При целочисленном делении значение `a / b` всегда представляет собой целое число в диапазоне от `-a` до `a`. Проверять на переполнение и потерю значимости необходимо, только если отрезок `[-a,a]` не полностью находится внутри отрезка `[MIN, MAX]`. Возвращаясь к нашему примеру с четырехбитным знаковым целым числом, в котором `MIN = -8`, а `MAX = 7`, видим, что единственный случай переполнения при делении `-8 / -1` (поскольку отрезок `[-8,8]` не полностью находится внутри отрезка `[-8,7]`). Фактически единственный сценарий переполнения при делении для знаковых целых чисел — когда `a` равно минимальному представимому

значению, а `b = -1`. При делении беззнаковых целых чисел переполнение вообще невозможно.

В табл. 2.1 и 2.2 подытожены этапы проверки на переполнение и потерю значимости для случаев, когда необходима особая обработка.

Таблица 2.1. Обнаружение целочисленного переполнения для a и b в диапазоне [MIN, MAX] при MIN = –MAX – 1

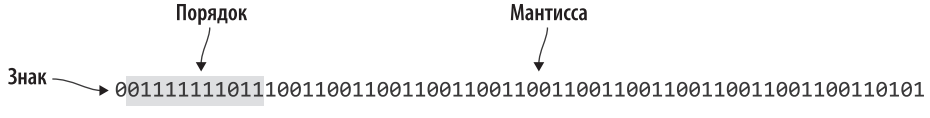
Сложение	Вычитание	Умножение	Деление
b > 0 и a > MAX – b	b < 0 и a > MAX + b	b > 0, a > 0 и a > MAX / b b < 0, a < 0 и a < MAX / b	a == MIN и b == –1

Таблица 2.2. Обнаружение целочисленной потери значимости для a и b в диапазоне [MIN, MAX] при MIN = –MAX – 1

Сложение	Вычитание	Умножение	Деление
b < 0 и a < MIN – b	b > 0 и a < MIN + b	b > 0, a < 0 и a < MIN / b b < 0, a > 0 и a > MIN / b	N/A

2.3.2. Типы с плавающей точкой и округление

IEEE 754 представляет собой стандарт Института инженеров электротехники и электроники для представления чисел *с плавающей точкой* (floating-point), то есть чисел с дробной частью. В TypeScript (и JavaScript) числа представляются в виде 64-битных чисел с плавающей точкой с помощью кодирования *binary64*. Подробное описание этого представления приведено на рис. 2.5.



Двоичное представление числа с плавающей точкой (0,10)

$$(-1)^{знак}\left(1+\sum_{i=1}^{52} Мантисса_{52-i}\times 2^{-i}\right)\times 2^{Порядок-1023}$$

Формула преобразования двоичного представления в фактическое значение

0.10000000000000000555111512312 6

Фактическое значение (аппроксимация числа 0,10)

Рис. 2.5. Представление числа с плавающей точкой 0,10. Во-первых, тут можно видеть двоичное представление в оперативной памяти трех компонентов: бита знака, порядка и мантиссы. Ниже приведена формула преобразования двоичного представления в число. Наконец, вы видите результат приложения этой формулы: 0,10 аппроксимируется значением 0,10000000000000000555111512312 6

Три компоненты, составляющие число с плавающей точкой: знак, порядок и мантисса. *Знак* (sign) — один бит со значением `0` для положительных чисел и `1` для отрицательных. *Мантисса* (mantissa) представляет собой дробную часть, как показано в формуле на рис. 2.5. Эта часть умножается на 2 в степени, соответствующей *смещенному порядку* (biased exponent).

Порядок называется *смещенным*, поскольку из представленного порядком беззнакового целого числа мы вычитаем определенное значение, чтобы он мог представлять как положительные, так и отрицательные числа. В случае кодирования binary64 это значение равно 1023. В стандарте IEEE 754 описано несколько кодировок, в ряде которых используется основание 10 вместо 2, хотя 2 в качестве основания на практике встречается чаще.

В стандарте также определено несколько специальных значений.

- `NaN` — расшифровывается как not a number («не число») и применяется для результата некорректных операций, например деления на 0.
- Положительная и отрицательная бесконечность (`Inf`), используемая в качестве максимальных (минимальных) значений при переполнении.
- И хотя согласно вышеприведенной формуле значение 0,10 превращается в число 0,10000000000000000555111512312 6, оно округляется до 0,1. На самом деле числа 0,10 и 0,10000000000000000555111512312 6 считаются в JavaScript равными. Единственная возможность представлять дробные числа из огромного диапазона значений при наличии относительно небольшого числа битов — с помощью округления и аппроксимации.

Точность

Если нужны точные значения — при работе с денежными суммами, например, — избегайте использования чисел с плавающей точкой. Дело в том, что суммирование 0,10 три раза не дает 0,30 ввиду того, что, хоть каждое отдельное представление 0,10 и округляется до 0,10, в результате их сложения получается число, которое округляется до 0,30000000000000004.

Небольшие целые числа можно спокойно представить без округления, так что лучше кодировать цену в виде двух целочисленных значений: одно для долларов, а второе для центов. В JavaScript есть функция `Number.isSafeInteger()`, позволяющая узнать, можно ли представить данное целочисленное значение без округления. На ее основе можно создать тип `Currency`, который кодирует два целочисленных значения и защищает от проблем округления, как показано в листинге 2.9.

В другом языке программирования мы воспользовались бы двумя переменными целочисленного типа и защитились от проблем переполнения/потери значимости. Но, поскольку в JavaScript нет простого целочисленного типа данных, мы применили функцию `Number.isSafeInteger()` для защиты от проблем округления. При работе с денежными суммами лучше выдать ошибку, чем обнаружить потом, что деньги появились/исчезли загадочным образом.

Листинг 2.9. Класс Currency и функция сложения денежных сумм

```
class Currency {
    private dollars: number;
    private cents: number;

    constructor(dollars: number, cents: number) {
        if (!Number.isSafeInteger(dollars))
            throw new Error("Cannot safely represent dollar amount");

        if (!Number.isSafeInteger(cents))
            throw new Error("Cannot safely represent cents amount");

        this.dollars = dollars;
        this.cents = cents;
    }

    getDollars(): number {
        return this.dollars;
    }

    getCents(): number {
        return this.cents;
    }
}

function add(currency1: Currency, currency2: Currency): Currency {
    return new Currency(
        currency1.getDollars() + currency2.getDollars(),
        currency1.getCents() + currency2.getCents());
}
```

Количество долларов и центов хранится в отдельных переменных

Конструктор класса гарантирует хранение только тех значений, которые можно безопасно представить без округления

Количества долларов и центов доступны через функции-геттеры, так что внешний код не может их модифицировать

Два значения Currency складываются просто путем сложения по отдельности количеств долларов и центов

Класс в листинге 2.9 — лишь каркас. Хорошим упражнением будет его расширение так, чтобы по достижении 100 в переменной для числа центов они автоматически превращались в доллар. Будьте осторожнее с проверкой безопасности целых чисел: что, если количество долларов представляет собой безопасное число, но при добавлении к нему 1 (получившейся из 100 центов) перестает быть таковым?

Сравнение чисел с плавающей точкой

Как мы видели, из-за округления обычно не имеет смысла проверять на равенство числа с плавающей точкой. Существует лучший способ выяснить, равны ли приблизительно два числа: проверить, не превышает ли их разность заданного порогового значения.

Какое пороговое значение следует выбрать? Оно должно равняться максимальной возможной погрешности округления. Это значение называется *машинным эпсилон* (machine epsilon) и зависит от способа кодирования. В JavaScript данное значение указано в константе `Number.EPSILON`. С его помощью можно реализовать проверку двух чисел на равенство, проверяя, не превышает ли абсолютное значение их разности

машинного эпсилон (листинг 2.10). Если нет, то эти значения отличаются друг от друга менее чем на погрешность округления, так что их можно считать равными.

Листинг 2.10. Равенство двух чисел с плавающей точкой в пределах эпсилон

```
function epsilonEqual(a: number, b: number): boolean {
    return Math.abs(a - b) <= Number.EPSILON;
}

console.log(0.1 + 0.1 + 0.1 == 0.3);
console.log(epsilonEqual(0.1 + 0.1 + 0.1, 0.3));
```

Проверяем, не превышает ли абсолютное значение разности двух чисел погрешности округления

Выводится false, поскольку 0,1 + 0,1 + 0,1 округляется до 0,30000000000000004

Выводится true, так как 0,3 и 0,30000000000000004 находятся не далее погрешности округления друг от друга

Обычно имеет смысл использовать какой-либо аналог функции `epsilonEqual` при сравнении чисел с плавающей точкой, поскольку арифметические операции могут вызывать ошибки округления, приводящие к неожиданным результатам.

2.3.3. Произвольно большие числа

В большинстве языков программирования есть библиотеки, позволяющие работать со сколь угодно большими числами. Данные типы способны увеличивать ширину до количества битов, требуемого для представления любого значения. В Python подобный тип является числовым типом по умолчанию, а для стандарта JavaScript сейчас предлагается использовать тип произвольно больших чисел `BigInt`. Тем не менее произвольно большие числа нельзя считать простыми типами данных, поскольку их можно построить на основе числовых типов фиксированной ширины. Они удобны, но во многих средах выполнения отсутствует их нативная реализация из-за отсутствия аппаратного эквивалента (микросхемы всегда работают с фиксированным количеством битов).

2.3.4. Упражнения

1. Что выведет следующий код?

```
let a: number = 0.3;
let b: number = 0.9;

console.log(a * 3 == b);
```

   А. Ничего; вернет ошибку.

   Б. `true`.

   В. `false`.

   Г. `0.9`.

2. Каким должно быть поведение при переполнении числа, служащего для отслеживания уникальных идентификаторов?
   А. Останов на максимальном значении.
   Б. Возврат на ноль.
   В. Возврат ошибки.
   Г. Подходит любой из предыдущих вариантов.

2.4. Кодирование текста

Еще один распространенный простой тип данных — *строка* (string), используемая для представления текста. Строка (строковое значение) состоит из нуля или более символов, так что это первый из описанных нами простых типов данных, который потенциально может принимать бесконечное множество значений.

На заре эпохи компьютеров для кодирования каждого символа использовался один байт, поэтому компьютеры могли представлять текст с помощью всего 256 символов. После введения стандарта Unicode, предназначенного для представления всех мировых алфавитов и других символов (таких как эмодзи), 256 символов явно стало недостаточно. На самом деле в Unicode описано более миллиона символов!

2.4.1. Разбиение текста

Рассмотрим пример простой функции разбиения текста, принимающей на входе строку и разбивающей ее на несколько строк заданной длины, которые поместились бы в окне текстового редактора, как показано в листинге 2.11.

Листинг 2.11. Простая функция разбиения текста

```
function lineBreak(text: string, lineLength: number): string[] {
    let lines: string[] = [];

    while (text.length > lineLength) {
        lines.push(text.substr(0, lineLength));
        text = text.substr(lineLength);
    }

    lines.push(text);
    return lines;
}
```

В массиве lines будет содержаться разбитый текст

Цикл повторяется, пока длина текста превышает заданную длину строки

Добавляем первые lineLength в качестве новой строки, после чего удаляем их из переменной text

Добавляем оставшийся текст (длина которого меньше lineLength) в итоговый результат в качестве последней строки

На первый взгляд, данная реализация должна работать корректно. Для входного текста `"Testing, testing"` и длины строки `5` получаются строки `["Testi", "ng, t",`

`"estin", g"]`. Именно этого мы и ожидали: текст разбивается на несколько строк на каждом пятом символе.

Но другие символы кодируются более сложным образом. Возьмем, например, эмодзи «женщина-полицейский»: 👮‍♀️. Хотя он и выглядит отдельным символом, JavaScript использует для его представления пять символов. Вызов `"👮‍♀️".length` возвращает `5`. Попытка разбить строку, содержащую такой эмодзи, на основе его внешнего вида в тексте может повлечь неожиданные результаты. Например, если разбить текст `...👮‍♀️` с длиной строки `5`, мы получим в качестве результата массив `["...👮", "♀️"]`.

Эмодзи «женщина-полицейский» состоит из двух отдельных эмодзи: «полицейский» и знак, обозначающий женский пол. Они объединяются с помощью соединительного символа нулевой длины `"\ud002"`. Он не имеет визуального представления и служит для объединения других символов.

Эмодзи «полицейский», 👮, представлен с помощью двух смежных символов, как видно при попытке разбиения более длинной строки `...👮‍♀️` с длиной строки `5`. В результате этого эмодзи «женщина-полицейский» разбивается на два и мы получаем `["....\ud83d", "\udc6e♀️"]`. Элемент `\uXXXX` — управляющие последовательности Unicode, представляющие символы, которые нельзя вывести в консоль «как есть». Эмодзи «женщина-полицейский», хоть и визуализируется как один символ, состоит из пяти различных управляющих последовательностей: `\ud83d`, `\udc6e`, `\u200d`, `\u2640` и `\ufe0e`.

Бездумное разбиение текста по границам символов может привести к невизуализируемым результатам и даже изменить смысл текста.

2.4.2. Кодировки

Чтобы разобраться, как правильно обрабатывать текст, необходимо изучить кодировки символов. Стандарт Unicode работает с двумя близкими, однако не идентичными понятиями: символы и графемы. *Символы* (characters) служат для представления текста (эмодзи «женщина-полицейский», объединяющий символ нулевой длины) в компьютере, а *графемы* (graphemes) — символы, которые видит пользователь (женщина-полицейский). При визуализации текста мы работаем с графемами, и разбивать многосимвольную графему нежелательно. В момент кодирования текста мы взаимодействуем с символами.

ГЛИФЫ И ГРАФЕМЫ

Глиф (glyph) — это конкретное представление символа. С полужирным шрифтом и С курсивом — две различные визуализации данного символа.

Графема (grapheme) — неделимая единица, которая теряет смысл при разбиении на составные части, как в примере с женщиной-полицейским. Графему можно представить с помощью нескольких глифов. Эмодзи Apple для женщины-полицейского внешне отличается от эмодзи Microsoft; они представляют собой различные глифы, визуализирующие одну графему (рис. 2.6).

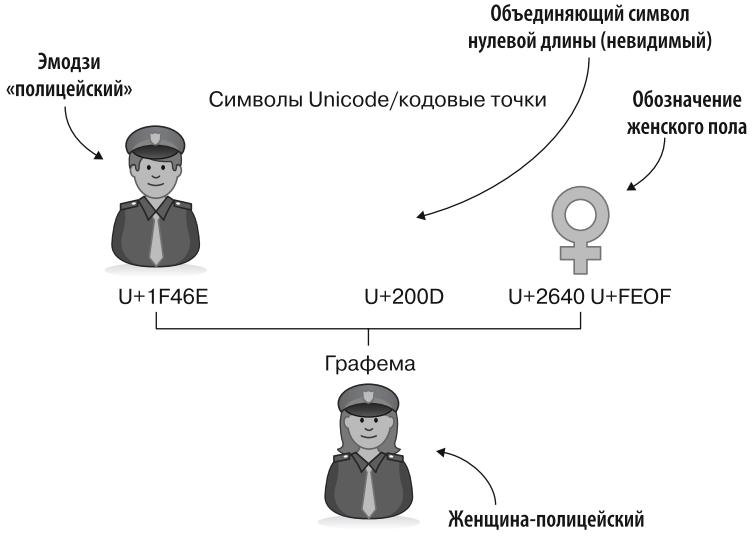


Рис. 2.6. Символьная кодировка эмодзи «женщина-полицейский» (символ эмодзи «женщина-полицейский» + объединяющий символ нулевой длины + эмодзи «женский пол») и графема, которая получается в результате (женщина-полицейский)

Каждый из символов Unicode описывается в виде кодовой точки, представляющей собой значение от `0x0` до `0x10FFFF`, так что всего существует 1 114 111[1] кодовых точек. Они отражают все алфавиты мира, эмодзи и множество других символов, и остается еще немало места для будущих дополнений.

UTF-32

Самая простая кодировка этих кодовых точек — UTF-32, в которой используется 32 бита для каждого символа; 32-битное целое число может представлять значения от `0x0` до `0xFFFFFFFF`, так что в нем поместится любая кодовая точка и останется немало незанятых чисел. Проблема кодировки UTF-32 состоит в ее крайне низкой эффективности, поскольку теряется очень много места для неиспользуемых битов. Как следствие, было разработано несколько более сжатых кодировок, применяющих меньше битов для первых кодовых точек и больше битов по мере роста значений. Их называют *кодировками переменной длины* (variable-length encodings).

UTF-16 и UTF-8

Чаще всего используются кодировки UTF-16 и UTF-8. В JavaScript применяется кодировка UTF-16. Единица кодировки в ней составляет 16 бит. Кодовые точки, которые умещаются в это количество битов (`0x0` до `0xFFFF`), представляются с помощью

[1] Точнее, 1114112. — *Примеч. пер.*

одного 16-битного целого числа, а кодовые точки, требующие более 16 бит (от `0x10000` до `0x10FFFF`), представляются с помощью двух 16-битных значений.

UTF-8, наиболее широко используемая кодировка, развивает этот подход: единица кодировки составляет 8 бит и кодовые точки представляются с помощью одного, двух, трех или четырех восьмибитных значений.

2.4.3. Библиотеки кодирования

Кодирование текста и выполнение над ним различных операций — сложная тема, которой посвящены целые книги. Хорошая новость: для эффективной работы со строками не нужно изучать все нюансы, но желательно осознавать всю сложность и искать возможности заменить бездумные операции над текстом, как в нашем примере с разбиением текста, вызовами функций из библиотек, инкапсулирующих эту сложность.

Например, библиотека `grapheme-splitter` для JavaScript предназначена для работы как с символами, так и с графемами. Установить ее можно с помощью команды `npm install grapheme-splitter`. Библиотека позволяет реализовать функцию `lineBreak()` для разбиения текста на уровне графем путем разбиения его на массив графем с последующей группировкой их в строки графем длиной `lineLength`, как показано в листинге 2.12.

Листинг 2.12. Функция для разбиения текста с помощью библиотеки grapheme-splitter

```
import GraphemeSplitter = require("grapheme-splitter");
const splitter = new GraphemeSplitter();

function lineBreak(text: string, lineLength: number) {
    let graphemes: string[] = splitter.splitGraphemes(text);
    let lines: string[] = [];

    for (let i = 0; i < graphemes.length; i += lineLength) {
        lines.push(graphemes.slice(i, i + lineLength).join(""));
    }
    return lines;
}
```

Функция splitGraphemes разбивает строку на массив графем

Получаем срезы графем длины lineLength и объединяем их в строки текста

При такой реализации строки `...`👮 и `....`👮 вообще не будут разбиваться при длине строки в `5`, поскольку ни одна из них не превышает длины пять графем, а строка `.....`👮 будет правильно разбита на `[".....","`👮`"]`.

Библиотека `grapheme-splitter` помогает предотвратить один из трех классов ошибок, часто встречающихся при работе со строками.

- *Выполнение операций над кодированным текстом на уровне символов, а не графем.* Мы рассмотрели данный пример в подразделе 2.4.1, где разбивали текст посимвольно, хотя для визуализации нам нужно было разбивать его по графемам. Попадание точки разбиения на пятый символ может привести к разбиению графемы на несколько отдельных графем. При отображении текста необходимо также учитывать то, из каких последовательностей символов состоят графемы.

- *Выполнение операций над кодированным текстом на уровне байтов, а не символов.* Подобная ситуация возможна, когда последовательность текста в кодировке переменной длины обрабатывается некорректно без учета кодировки, в результате чего символ может оказаться разбит на несколько, например при разбиении по пятому байту, когда нужно было разбивать по пятому символу. В зависимости от кодировки конкретный символ может содержать один байт или более, так что допущения, которые не учитывают способ кодирования, нежелательны.
- *Интерпретация последовательности байтов как текста с неправильной кодировкой* (например, пытаться интерпретировать текст в кодировке UTF-16 как текст в UTF-8, и наоборот). Необходимо знать, какова кодировка текста, полученного от другого компонента в виде байтовой последовательности. В различных языках приняты разные кодировки для текста по умолчанию, поэтому нельзя просто интерпретировать байтовые последовательности как строки — можно получить неправильную интерпретацию.

На рис. 2.7 показано, что графема «женщина-полицейский» состоит из двух символов Unicode. На этом рисунке также приведены их коды UTF-16 и бинарное представление.

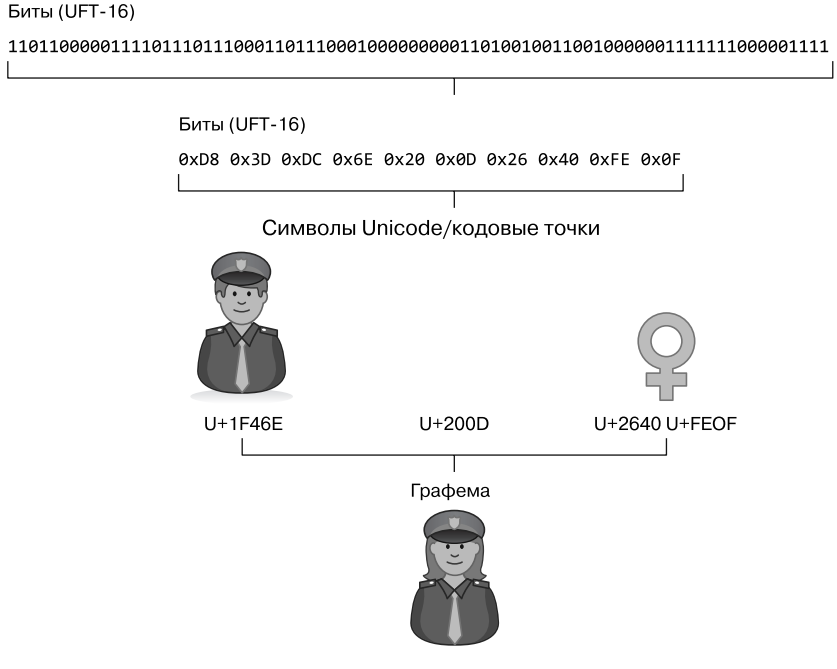


Рис. 2.7. Эмодзи «женщина-полицейский» в виде битов в памяти в строковой кодировке UTF-16, байтовой последовательности UTF-16, последовательности кодовых точек UTF-16 и графемы

Обратите внимание, что UTF-8-кодирование для этой же графемы отличается, хотя экранное представление такое же. Кодирование UTF-8 для нее: `0xF0 0x9F 0x91 0xAE 0xE2 0x80 0x8D 0xE2 0x99 0x80 0xEF 0xB8 0x8F`.

Всегда проверяйте правильность кодировки, на основе которой вы интерпретируете последовательности байтов, и используйте строковые библиотеки для операций со строками на уровне символов и графем.

2.4.4. Упражнения

1. Сколько байтов необходимо для кодирования символа UTF-8?
   А. 1 байт.
   Б. 2 байта.
   В. 4 байта.
   Г. Зависит от символа.
2. Сколько байтов необходимо для кодирования символа UTF-32?
   А. 1 байт.
   Б. 2 байта.
   В. 4 байта.
   Г. Зависит от символа.

2.5. Создание структур данных на основе массивов и ссылок

Последние два простых типа данных, которые мы обсудим, — массивы и ссылки. С их помощью можно создать любую более сложную структуру данных, например список или дерево. У реализации структур данных на основе каждого из этих двух простых типов есть свои плюсы и минусы. Мы обсудим подробнее, как лучше их использовать в зависимости от ожидаемых паттернов обращения (частота чтения относительно частоты записи) и плотности данных (плотные или разреженные).

В массиве фиксированной длины хранится несколько значений определенного типа, одно за другим, что обеспечивает эффективный доступ. Ссылочные же типы позволяют разбивать структуру данных по нескольким местам благодаря ссылкам одних компонентов на другие.

Мы не относим массивы переменной длины к простым типам данных, поскольку они реализуются на основе массивов фиксированной длины и/или ссылок, как мы увидим в этом разделе.

2.5.1. Массивы фиксированной длины

Массивы фиксированной длины представляют непрерывную область оперативной памяти, содержащую несколько значений одного типа. Так, массив из пяти 32-битных целых чисел занимает область 160 бит (5×32), в котором в первых 32 битах содержится первое число, во вторых 32 битах — второе и т. д.

Массивы используются чаще, чем, скажем, связные списки, из соображений быстродействия: доступ к любому из хранящихся последовательно значений не требует много времени. Если массив 32-битных целых чисел начинается по адресу 101 в оперативной памяти (первое целое число (с индексом 0) хранится в виде 32 бит, от 101 до 132), то целое число с индексом N в массиве находится по адресу $101 + N \times 32$. В общем случае, если массив начинается по адресу *base*, а размер элемента M, то элемент с индексом N находится по адресу $base + N \times M$. Поскольку оперативная память непрерывна, достаточно высока вероятность попадания массива в одну страницу памяти и кэширования целиком, что позволит обращаться к нему очень быстро.

Напротив, для доступа к N-му элементу связного списка придется начать с головы (head) списка и переходить по указателям `next` узлов, пока не будет достигнут N-й элемент. Вычислить адрес узла напрямую невозможно. Память под узлы не обязательно выделяется последовательно, так что может понадобиться подгрузить/выгрузить несколько страниц памяти, прежде чем будет достигнут желаемый узел. На рис. 2.8 приведены представления массивов и связных списков целых чисел в оперативной памяти.

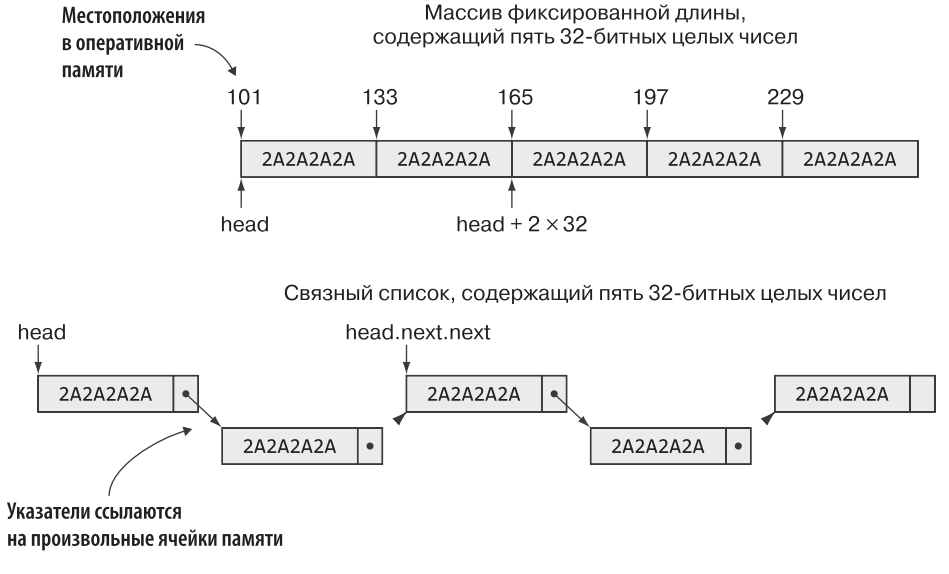


Рис. 2.8. Хранение пяти 32-битных целых чисел в массиве фиксированной длины и в связном списке. Поиск элемента в таком массиве производится чрезвычайно быстро, поскольку можно вычислить по формуле его точное местоположение. Напротив, при работе со связным списком необходимо следовать по указателям next элементов списка вплоть до достижения нужного элемента. Элементы могут располагаться в любом месте оперативной памяти

Термин «фиксированная длина» (fixed-size) означает, что массив нельзя увеличить в размере или сжать. Если понадобится сохранить в массиве шестой элемент, то придется выделить память под новый массив, вмещающий шесть элементов,

и скопировать первые пять из старого. А в связный список, в отличие от массива, можно добавить узел без каких-либо модификаций уже существующих узлов. В зависимости от предполагаемого паттерна доступа (больше операций чтения или операций записи) лучше подойдет либо первое представление, либо второе.

2.5.2. Ссылки

Ссылочные типы содержат указатели на объекты. Значение ссылочного типа — содержащиеся в переменной биты — отражает не содержимое объекта, а лишь место, где он находится. Несколько ссылок на один объект не означают дублирования состояния объекта, поэтому изменения такого объекта, произведенные через одну из ссылок, видны через все остальные ссылки.

Ссылочные типы часто используются в реализациях структур данных, поскольку позволяют связывать отдельные компоненты, добавлять их в структуру данных во время выполнения и удалять из нее.

Ниже мы рассмотрим несколько распространенных структур данных и узнаем, как их реализовать с помощью массивов и ссылок или путем их сочетания.

2.5.3. Эффективная реализация списков

В стандартной библиотеке многих языков программирования существует реализация структуры данных *список*. Обратите внимание: это не простой тип данных, а структура данных, реализованная на основе простых типов данных. Списки могут сжиматься/расти по мере удаления/добавления элементов.

Реализация списков в виде связных позволяет добавлять и удалять узлы без копирования данных, но обход списка оказывается весьма дорогостоящим (линейное время обхода, то есть сложность порядка `O(n)`, где `n` — длина списка). В листинге 2.13 приведена подобная реализация списка — `NumberLinkedList` с двумя функциями: `at()`, для извлечения значения элемента списка с заданным индексом, и `append()`, добавляющая значение в конец списка. Эта реализация содержит две ссылки: одну на начало списка, с которого можно начинать обход, и вторую — на конец списка. Благодаря ей можно добавлять новые элементы, не обходя весь список.

Листинг 2.13. Реализация связного списка

```
class NumberListNode {
    value: number;
    next: NumberListNode | undefined;

    constructor(value: number) {
        this.value = value;
        this.next = undefined;
    }
}

class NumberLinkedList {
    private tail: NumberListNode = {value: 0, next: undefined };
    private head: NumberListNode = this.tail;
```

Каждый узел списка содержит значение и ссылку на следующий узел (или undefined, если это последний узел)

Сначала создается пустой список, в котором как головной, так и хвостовой элементы указывают на фиктивные узлы

```
    at(index: number): number {
        let result: NumberListNode | undefined = this.head.next;
        while (index > 0 && result != undefined) {
            result = result.next;
            index--;
        }

        if (result == undefined) throw new RangeError();

        return result.value;
    }

    append(value: number) {
        this.tail.next = {value: value, next: undefined };
        this.tail = this.tail.next;
    }
}
```

Для получения узла с заданным индексом необходимо начать с головы списка и следовать по ссылкам next

Узел добавляется достаточно эффективным образом: мы просто дописываем его в хвост списка, после чего обновляем значение свойства tail

Как видим, функция `append()` в данном случае реализована очень эффективно, поскольку должна лишь добавить узел в хвост списка и сделать этот новый узел хвостовым. С другой стороны, функция `at()` начинает с головы списка и проходит по ссылкам `next` вплоть до достижения искомого узла.

В листинге 2.14 мы сравним это с реализацией на основе массива, в которой доступ к элементу производится эффективно, а вот добавление элемента является дорогостоящей операцией.

Листинг 2.14. Реализация списка на основе массива

```
class NumberArrayList {
    private numbers: number[] = [];
    private length: number = 0;

    at(index: number): number {
        if (index >= this.length) throw new RangeError();
        return this.numbers[index];
    }

    append(value: number) {
        let newNumbers: number[] = new Array(this.length + 1);
        for (let i = 0; i < this.length; i++) {
            newNumbers[i] = this.numbers[i];
        }
        newNumbers[this.length] = value;
        this.numbers = newNumbers;
        this.length++;
    }
}
```

Значения хранятся в массиве number изначально нулевой длины

Доступ к элементу производится просто путем выбора элемента из массива по индексу

Добавление числа в массив требует выделения памяти для нового массива и копирования старых элементов

Наконец, последний элемент добавляется в конец нового массива

Здесь доступ к элементу с заданным индексом означает просто выбор элемента из базового массива `number` по индексу. А вот добавление нового значения становится непростой операцией.

1. Необходимо выделить память под новый массив, на один элемент больше текущего.

2. Необходимо скопировать все элементы из текущего массива в новый.
3. Далее новое значение нужно добавить в новый массив в качестве последнего элемента.
4. Текущий массив следует заменить новым.

Копирование всех элементов массива при каждом добавлении нового элемента, скажем прямо, не самая эффективная реализация.

На практике большинство библиотек реализуют списки в виде массивов с некоторым запасом места. Выбирается больший размер массива, чем требуется изначально, поэтому новые элементы можно добавить, не прибегая к созданию нового массива и копированию данных. При заполнении массива выделяется память под новый массив, правда, двойного размера, и элементы копируются в него (рис. 2.9).

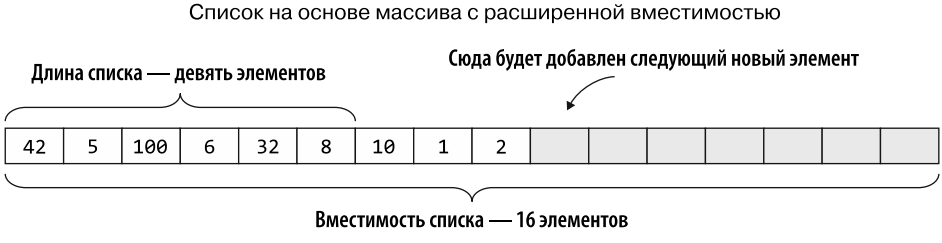


Рис. 2.9. Список на основе массива, содержащий девять элементов, но потенциально вмещающий 16. В него можно добавить еще семь элементов, прежде чем придется переносить данные в новый, больший массив

Благодаря такому эвристическому алгоритму вместимость массива растет экспоненциально, поэтому данные не приходится копировать так часто, как если бы массив наращивался по одному элементу за раз (листинг 2.15).

Листинг 2.15. Реализация списка на основе массива с расширенной вместимостью

```
class NumberList {
    private numbers: number[] = new Array(1);
    private length: number = 0;
    private capacity: number = 1;

    at(index: number): number {
        if (index >= this.length) throw new RangeError();
        return this.numbers[index];
    }

    append(value: number) {
        if (this.length < this.capacity) {
            this.numbers[length] = value;
            this.length++;
            return;
        }
```

Хотя список пуст, мы начинаем с вместимости 1

Доступ к элементам производится аналогично предыдущей реализации

Если массив заполнен не полностью, то можно просто добавить элемент и обновить значение длины (length)

```
        this.capacity = this.capacity * 2;
        let newNumbers: number[] = new Array(this.capacity);
        for (let i = 0; i < this.length; i++) {
            newNumbers[i] = this.numbers[i];
        }
        newNumbers[this.length] = value;
        this.numbers = newNumbers;
        this.length++;
    }
}
```

При полном заполнении массива необходимо выделить память под новый и скопировать элементы, но при этом удвоить вместимость, чтобы при соответствующем количестве последующих добавлений элементов не потребовалось выделять память заново

Аналогично можно реализовать другие линейные структуры данных, например стеки и кучи. Эти структуры оптимизированы для доступа на чтение, который всегда чрезвычайно эффективен. Расширенная вместимость обеспечивает эффективность большинства операций записи, однако некоторые записи при полном заполнении структуры данных требуют переноса всех элементов в новый массив, что нерезультативно. Вдобавок при этом образуется перерасход памяти, поскольку список выделяет ее для большего количества элементов, чем требуется в настоящий момент, чтобы освободить место для будущих добавлений.

2.5.4. Бинарные деревья

Рассмотрим другой тип структуры данных: структуру, в которой можно добавлять элементы в различные места. Примером может служить бинарное дерево, в котором новый узел можно присоединить к любому другому, еще не имеющему двух дочерних узлов.

Один из вариантов: представить бинарное дерево с помощью массива. На первом уровне дерева, корневом, содержится не более одного узла. На втором — не более двух: дочерние узлы корневого. На третьем — не более четырех: дочерние узлы двух узлов предыдущего уровня и т. д. В общем случае у дерева с N уровнями может быть не более $1 + 2 + ... + 2^{N-1}$ узлов, что равняется $2^N - 1$.

Бинарное дерево можно хранить в массиве, расположив в нем уровни один за другим. Если дерево не полное (не на всех уровнях присутствуют все возможные узлы), то мы будем отмечать недостающие узлы `undefined`. Преимущество этого представления — легкость перехода от родительского узла к дочерним: если родительский узел располагается в массиве по индексу `i`, то левый дочерний узел будет находиться по индексу `2*i`, а правый — `2*i+1`.

На рис. 2.10 показано представление бинарного дерева с помощью массива фиксированной длины.

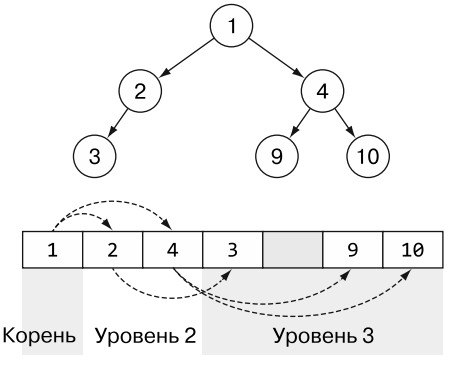


Рис. 2.10. Представление бинарного дерева с помощью массива фиксированной длины. Отсутствующий узел (правый дочерний узел узла 2) соответствует неиспользуемому элементу массива. Связь «предок — потомок» между узлами неявная, поскольку индекс дочернего узла можно вычислить на основе индекса родительского узла, и наоборот

Добавление узла также происходит достаточно эффективно, если не меняется количество уровней дерева. Однако при добавлении нового уровня необходимо не только скопировать все дерево, но и удвоить размер массива, чтобы хватило места для всех возможных узлов, как показано в листинге 2.16. Это аналогично эффективной реализации списка.

Листинг 2.16. Реализация бинарного дерева на основе массива

```
class Tree {
    nodes: (number | undefined)[] = [];

    left_child_index(index: number): number {
        return index * 2;
    }

    right_child_index(index: number): number {
        return index * 2 + 1;
    }

    add_level() {
        let newNodes: (number | undefined)[] =
            new Array(this.nodes.length * 2 + 1);

        for (let i = 0; i < this.nodes.length; i++) {
            newNodes[i] = this.nodes[i];
        }
        this.nodes = newNodes;
    }
}
```

Узлы хранятся в виде массива числовых значений и значений undefined (обозначающих пропуски)

Вычисление индексов левого и правого дочерних узлов по индексу родительского узла

Увеличение вместимости при добавлении нового уровня требует удвоения размера массива и переноса узлов

У этой реализации есть недостаток: в случае разреженных деревьев требуемый объем дополнительного пространства может оказаться неприемлемым (рис. 2.11).

Из-за избыточного расхода памяти для более компактного представления бинарных деревьев обычно используются ссылочные структуры данных (листинг 2.17). При этом в каждом узле хранятся значение и ссылки на дочерние узлы.

При такой реализации дерево представлено ссылкой на свой корневой узел. С этой отправной точки можно достичь любого узла дерева, следуя по левым/правым дочерним узлам. Для добавления узла в произвольном месте достаточно выделить под него память и задать значение свойства `left` или `right` его родительского узла. На рис. 2.12 показано представление разреженного дерева с помощью ссылок.

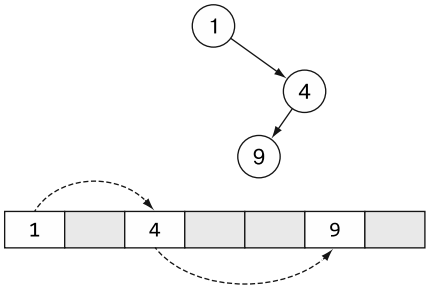


Рис. 2.11. Для корректного представления разреженного бинарного дерева, содержащего всего три узла, тем не менее требуется массив из семи элементов. А если у узла 9 появится дочерний узел, то размер массива вырастет до 15

Листинг 2.17. Компактная реализация бинарного дерева

```
class TreeNode {
    value: number;
    left: TreeNode | undefined;
    right: TreeNode | undefined;

    constructor(value: number) {
        this.value = value;
        this.left = undefined;
        this.right = undefined;
    }
}
```

В каждом узле хранится значение

Поля left и right ссылаются на другие узлы или содержат значение undefined, если у данного узла нет дочерних

Хотя для самих ссылок нужно некое ненулевое количество памяти, требуемый объем пространства пропорционален количеству узлов. Для разреженных деревьев подобное представление подходит гораздо лучше, чем реализация на основе массива, при которой пространство растет экспоненциально с количеством уровней.

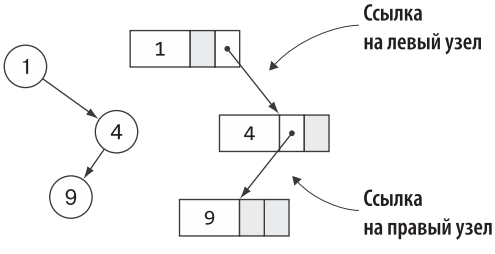


Рис. 2.12. Представление разреженного дерева с помощью ссылок. Схема справа демонстрирует структуру данных узла в виде значения, ссылки на левый и правый дочерний узлы

В общем случае для представления разреженных структур данных, в которых может быть множество «пропусков», а элементы могут добавляться в различные места, гораздо лучше подходит вариант, когда одни элементы ссылаются на другие. Размещение же всей структуры данных в массиве фиксированной длины чревато неприемлемыми накладными расходами.

2.5.5. Ассоциативные массивы

Некоторые языки программирования предоставляют встроенную поддержку синтаксиса и других простых типов структур данных. Один из часто встречающихся подобных типов — *ассоциативный массив* (associative array), также известный под названиями *«словарь»* (dictionary) и *«хеш-таблица»* (hash table). Этот тип структуры данных представляет собой набор пар «ключ — значение» и обеспечивает эффективное извлечение значения по ключу.

Вопреки тому, что вы могли подумать при чтении предыдущих примеров, массивы JavaScript/TypeScript — ассоциативные. В этих языках программирования нет простого типа данных, соответствующего массиву фиксированной длины. Наши примеры кода демонстрируют, как можно реализовать структуры данных на основе массивов фиксированной длины. Массивы фиксированной длины предполагают чрезвычайно эффективный доступ по индексу и неизменяемый размер. В случае JavaScript/TypeScript этого нет. Мы рассматриваем тут массивы фиксированной длины вместо ассоциативных потому, что последние можно реализовать с помощью обычных

массивов и ссылок. Для наглядности мы рассматриваем массивы TypeScript как массивы фиксированной длины, чтобы примеры кода можно было непосредственно перенести на большинство других популярных языков программирования.

В таких языках программирования, как Java и C#, массивы и ссылки являются простыми типами данных, а словари и хеш-карты входят в стандартную библиотеку. В JavaScript и Python ассоциативные массивы — простые типы данных, но среды выполнения также реализуют их на основе массивов и ссылок. Массивы и ссылки — низкоуровневые конструкции, отражающие определенные схемы размещения данных в оперативной памяти и модели доступа, в то время как ассоциативные массивы являются высокоуровневыми абстракциями.

Ассоциативные массивы часто реализуются как массивы фиксированной длины, элементами которых являются списки. Хеш-функция принимает на входе ключ произвольного типа и возвращает индекс в массиве фиксированной длины. Пара «ключ — значение» добавляется в список или извлекается из него по заданному индексу в массиве. Списки используются потому, что хеш нескольких ключей может соответствовать одному индексу (рис. 2.13).

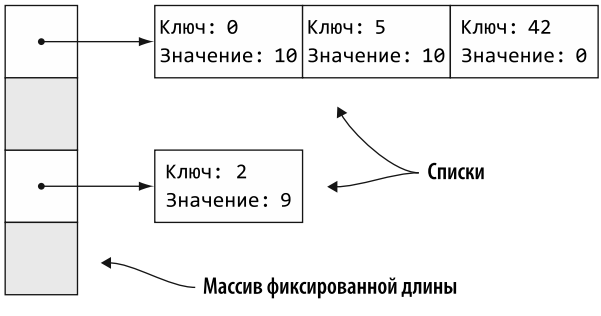


Рис. 2.13. Реализация ассоциативного массива в виде массива списков. Данный экземпляр содержит соответствия «ключ — значение»: 0 → 10, 2 → 9, 5 → 10 и 42 → 0

Поиск значения по ключу включает поиск списка, в котором находится пара «ключ — значение», обход его для нахождения нужного ключа и возврат значения. Если список слишком длинный, то время поиска возрастает, так что эффективные реализации ассоциативных массивов производят перебалансировку с увеличением размера массива, уменьшая за счет этого размеры списков.

Хорошая функция хеширования обеспечивает равномерное распределение ключей по спискам, чтобы длины списков были примерно равны.

2.5.6. Соотношения выгод и потерь различных реализаций

В предыдущем подразделе мы увидели, что массивов и ссылок вполне достаточно для реализации других структур данных. В зависимости от ожидаемых паттернов обращения (например, частоты чтения относительно частоты записи) и формы данных (плотные или разреженные) можно подобрать нужные простые типы для

компонентов структуры данных и объединить их так, чтобы получилась наиболее эффективная реализация.

Чтение/обновление в массивах фиксированной длины происходит чрезвычайно быстро, они отлично подходят для представления плотных данных. Что касается структур данных переменного размера, ссылки позволяют эффективнее добавлять новые данные и лучше подходят для представления разреженных данных.

2.5.7. Упражнение

Какая структура данных лучше подойдет для обращения к элементам в случайном порядке?

А. Связный список.

Б. Массив.

В. Словарь.

Г. Очередь.

Резюме

- Функции, которые никогда ничего не возвращают (работают бесконечно или генерируют исключения), следует объявлять как возвращающие пустой тип. Пустой тип можно реализовать в виде класса, не допускающего создания экземпляров, или как перечисляемый тип, не содержащий элементов.
- Функции, которые не возвращают никакого осмысленного результата по завершении выполнения, следует объявлять как возвращающие единичный тип (в большинстве языков программирования — `void`). Единичный тип можно реализовать в виде класса-одиночки или перечисляемого типа, содержащего один элемент.
- Вычисление булевых выражений обычно выполняется по сокращенной схеме, поэтому на то, какие из операндов будут вычислены, влияет их порядок.
- Возможно переполнение целочисленных типов фиксированной ширины. Поведение по умолчанию при переполнении зависит от языка программирования. А желаемое поведение зависит от конкретного сценария использования.
- Представление чисел с плавающей точкой — приближенное, так что лучше не сравнивать значения на равенство, а проверять, не отстоят ли они дальше `EPSILON` друг от друга.
- Текст состоит из графем, представление которых строится из одной или нескольких кодовых точек Unicode; каждая из них кодируется одним байтом или более. Библиотеки для операций над строками ограждают нас от всех сложностей кодирования и представления строк, так что лучше полагаться на них, а не производить операции над текстом напрямую.
- Массивы фиксированной длины и ссылки — стандартные «строительные блоки» структур данных. В зависимости от паттернов обращения к данным и степени их плотности можно использовать те или другие либо их сочетание для эффективной реализации любой, сколь угодно сложной структуры данных.

Ответы к упражнениям

2.1. Проектирование функций, не возвращающих значений

1. В — функция `set()` не возвращает никакого осмысленного значения, так что единичный тип `void` отлично подойдет в качестве возвращаемого типа.
2. А — функция `set()` никогда ничего не возвращает, поэтому в качестве возвращаемого типа отлично подойдет пустой тип `never`.

2.2. Булева логика и сокращенные схемы вычисления

Б — значение счетчика увеличивается только один раз, поскольку функция возвращает `false`, так что булево выражение вычисляется по сокращенной схеме.

2.3. Распространенные ловушки числовых типов данных

1. В — из-за округления чисел с плавающей точкой результат вычисления выражения — `false`.
2. В — оптимальным поведением в данном случае будет выдача ошибки, поскольку идентификаторы должны быть уникальными.

2.4. Кодирование текста

1. Г — UTF-8 — кодировка переменной длины.
2. В — UTF-32 — кодировка фиксированной длины; все символы кодируются четырьмя байтами.

2.5. Создание структур данных на основе массивов и ссылок

Б — для произвольного доступа лучше подходят массивы.

3 Составные типы данных

В этой главе

- Объединение типов в составные типы данных.
- Объединение типов в XOR-типы данных.
- Реализация паттерна проектирования «Посетитель».
- Алгебраические типы данных.

В главе 2 мы рассмотрели некоторые простые типы данных — строительные блоки системы типов. В текущей главе мы обсудим способы их сочетания в целях описания новых типов данных.

Мы рассмотрим составные типы данных, агрегирующие значения нескольких типов. Мы узнаем, как за счет правильного наименования членов классов придать осмысленность данным и снизить риск некорректной интерпретации, а также гарантировать соответствие значений определенным ограничениям.

Далее мы обсудим XOR-типы данных (either-or types) (строго дизъюнктивные типы), содержащие ровно одно значение одного или нескольких типов. Мы рассмотрим такие распространенные типы данных, как опционалы, XOR-типы данных и вариантные типы данных, а также некоторые их приложения. Мы увидим, например, почему возвращать результат *или* ошибку обычно безопаснее, чем возвращать результат *и* ошибку.

В качестве приложения XOR-типов данных мы рассмотрим паттерн проектирования «Посетитель» и сравним реализацию, использующую иерархии классов,

с реализацией, в которой для хранения объектов и выполнения операций над ними используется вариантный тип данных.

И наконец, вас ждет описание алгебраических типов данных (ADT) и их связи с вопросами, обсуждаемыми в этой главе.

3.1. Составные типы данных

Простейший способ сочетания типов данных — группировка их в новые типы. Возьмем пару координат x и y на плоскости. Тип обеих координат x и y — `number`. У точки на плоскости есть обе координаты (x и y), поэтому два типа в ней объединяются в третий, значениями которого служат пары чисел.

В общем случае объединение одного или нескольких типов подобным образом приводит к созданию нового типа данных, значениями которого являются все возможные сочетания составляющих его типов (рис. 3.1).

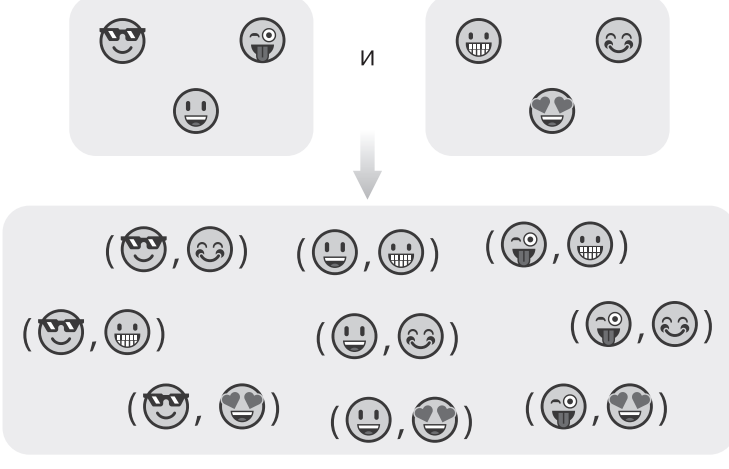


Рис. 3.1. Объединение двух типов таким образом, чтобы итоговый содержал по одному значению из каждого этого типа. Каждый эмодзи представляет значение одного из этих типов. Скобки отражают тот факт, что значения объединенного типа являются парами значений исходных типов

Обратите внимание: речь идет про объединение значений типов, а не операций над ними. Комбинирование операций мы обсудим, когда будем рассматривать элементы объектно-ориентированного программирования в главе 8. Пока же ограничимся значениями.

3.1.1. Кортежи

Допустим, нам нужно вычислить расстояние между двумя точками, заданными в виде пар координат. Можно определить функцию, которая принимает координаты x и y сначала первой точки, а затем второй и вычисляет расстояние между ними, как показано в листинге 3.1.

Листинг 3.1. Расстояние между двумя точками

```
function distance(x1: number, y1: number, x2: number, y2: number)
    : number {
    return Math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2);
}
```

Этот код работает, но не идеален: координата `x1`, если речь идет о точках, не имеет смысла без соответствующей координаты `y1`. Скорее всего, нужно будет производить операции над точками во многих местах нашего приложения. Поэтому вместо того, чтобы передавать отдельно координаты x и y, мы можем сгруппировать их в *кортеж* (tuple).

ТИПЫ-КОРТЕЖИ

Тип-кортеж состоит из набора типов-компонентов, к которым можно обращаться по их позициям в кортеже. Кортежи — способ группировки данных специально для конкретного случая, позволяющий передавать в виде одной переменной несколько значений различных типов.

С помощью кортежей можно передавать пары координат x и y как цельные точки. Это упрощает чтение и написание кода. Чтение — поскольку теперь понятно, что мы имеем дело с точками, а написание — поскольку можно использовать объявление `point: Point` вместо `x: number, y: number`, как показано в листинге 3.2.

Листинг 3.2. Расстояние между двумя точками, описанными в виде кортежей

```
type Point = [number, number];  ◄── Описываем новый тип данных — кортеж чисел

function distance(point1: Point, point2: Point): number {
    return Math.sqrt(
        (point1[0] - point2[0]) ** 2 + (point1[1] - point2[1]) ** 2);
}
```

Кортежи удобны также для возврата из функции нескольких значений, что сложно сделать без группировки значений. Либо можно использовать параметры `out` — обновляемые функцией аргументы, которые, впрочем, затрудняют анализ кода.

Кортеж своими руками

В большинстве языков программирования существует встроенный синтаксис для кортежей или же кортежи входят в стандартную библиотеку. Тем не менее рассмотрим, как можно реализовать кортеж, если он отсутствует. В листинге 3.3 мы реализуем обобщенный кортеж, включающий два типа-компонента, который называют также *парой* (pair).

Рассматривая типы как множества возможных значений, можно сказать, что если координаты x и y могут принимать любые значения из задаваемого типом `number` множества, то кортеж `Point` может принимать любое значение из задаваемого парой `<number, number>` множества пар.

Листинг 3.3. Тип Pair

```
class Pair<T1, T2> {
    m0: T1;
    m1: T2;

    constructor(m0: T1, m1: T2) {
        this.m0 = m0;
        this.m1 = m1;
    }
}

type Point = Pair<number, number>;

function distance(point1: Point, point2: Point): number {
    return Math.sqrt(
        (point1.m0 - point2.m0) ** 2 + (point1.m1 - point2.m1) ** 2);
}
```

Тип Pair включает значения типов T1 и T2

3.1.2. Указание смыслового содержания

Точки вполне можно описывать как пары чисел, но при этом теряется определенная часть смыслового содержания: пара чисел интерпретируется либо как координаты x и y, либо как координаты y и x (рис. 3.2).

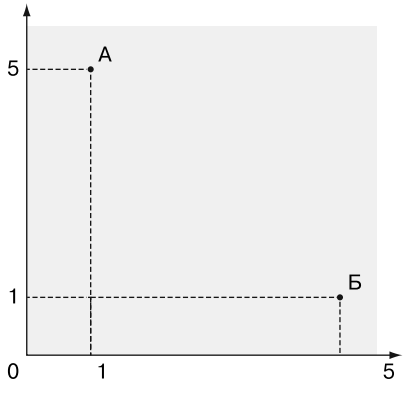


Рис. 3.2. Два способа интерпретации пары (1, 5): как точка А с координатой x = 1 и координатой y = 5, либо как точка Б с координатой x = 5 и координатой y = 1

До сих пор в наших примерах предполагалось, что первый компонент — координата x, а второй — y. Данное допущение работает, но оставляет возможности для ошибок. Лучше было бы закодировать смысл в саму систему типов, таким образом гарантируя невозможность неправильной интерпретации x как y или y как x. Сделать это можно с помощью так называемого *типа-записи* (record type).

ТИПЫ-ЗАПИСИ

Типы-записи аналогично кортежам объединяют значения нескольких других типов. Но вместо того, чтобы обращаться к значениям компонентов в соответствии с их позицией в кортеже, типы-записи позволяют давать компонентам названия и обращаться по ним. Типы-записи в различных языках называются record («запись») или struct («структура»).

Если описать наш тип `Point` как структуру, то можно будет назначить для двух ее компонентов названия `x` и `y` и исключить всякую неоднозначность, как видно из листинга 3.4.

Листинг 3.4. Расстояние между двумя точками, описанными как записи

```
class Point {
    x: number;
    y: number;

    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
}

function distance(point1: Point, point2: Point): number {
    return Math.sqrt(
        (point1.x - point2.x) ** 2 + (point1.y - point2.y) ** 2);
}
```

В классе Point определены члены класса x и y, благодаря чему понятно, какой координате соответствует тот или иной компонент

В качестве эмпирического правила можно порекомендовать описывать записи с поименованными компонентами вместо передачи кортежей. Отсутствие названий в кортежах у компонентов порождает возможность неправильной интерпретации. Кортежи обычно ничем не лучше записей в смысле быстродействия или функциональности, за исключением того, что обычно при использовании объявляются как встроенные, а для записей требуется отдельное определение. В большинстве случаев как раз имеет смысл добавить отдельное описание, поскольку оно придает переменным дополнительный смысл.

3.1.3. Сохранение инвариантов

В языках программирования, где у типов-записей могут быть методы, обычно есть и возможность описания видимости членов этих типов. Член такого типа может быть публичным (`public`) — доступным из любого места кода, приватным (`private`) — доступным только в пределах записи и т. д. В TypeScript члены классов по умолчанию публичны.

В общем случае при описании типов-записей, если члены типа не зависят друг от друга и их изменение не приводит к проблемам, их можно смело описывать как публичные. Как раз такая ситуация и имеет место при описании точек как пар координат x и y: координаты могут меняться независимо друг от друга при перемещении точки по плоскости.

Рассмотрим другой пример, в котором члены типа не могут беспроблемно меняться независимо друг от друга: тип данных — для денежных сумм, о котором мы говорили в главе 2, состоящий из количества долларов и центов. Расширим описание этого типа следующими правилами, определяющими корректное количество денег.

- Количество долларов должно представлять собой неотрицательное целое число, подходящее для безопасного представления с помощью типа `number`.
- Количество центов должно представлять собой неотрицательное целое число, подходящее для безопасного представления с помощью типа `number`.
- Количество центов не должно превышать 99; каждые следующие 100 центов необходимо преобразовывать в 1 доллар.

Подобные правила, гарантирующие верную структуру значений, называются *инвариантами* (invariants), поскольку должны соблюдаться при изменении значений, входящих в составной тип данных (compound type). Если сделать члены типа публичными, то внешний код сможет их менять, в результате чего записи могут оказаться сформированы некорректно, как показано в листинге 3.5.

Листинг 3.5. Денежная сумма некорректного вида

```
class Currency {
    dollars: number;
    cents: number;

    constructor(dollars: number, cents: number) {
        if (!Number.isSafeInteger(cents) || cents < 0)
            throw new Error();

        dollars = dollars + Math.floor(cents / 100);
        cents = cents % 100;

        if (!Number.isSafeInteger(dollars) || dollars < 0)
            throw new Error();

        this.dollars = dollars;
        this.cents = cents;
    }
}

let amount: Currency = new Currency(5, 50);
amount.cents = 300;
```

Конструктор гарантирует, что значения для долларов и центов будут корректными

Каждые последующие 100 центов преобразуются в 1 доллар

К сожалению, публичность членов класса позволяет внешнему коду менять объект некорректным образом

Этой ситуации можно избежать, сделав члены класса приватными и описав методы, предназначенные для их изменения, которые обеспечивали бы соблюдение инвариантов, как показано в листинге 3.6. Обработка всех случаев, при которых инварианты нарушаются, гарантирует, что объект всегда будет находиться в допустимом состоянии, поскольку изменение его либо приведет к другому объекту корректного вида, либо вызовет генерацию исключения.

Теперь внешнему коду придется менять значения только через функции `assignDollars()` и `assignCents()`, что гарантирует сохранение инвариантов: в случае некорректности передаваемых значений генерируется исключение. Если количество центов превышает 100, то сотни центов преобразуются в доллары.

В целом при отсутствии необходимости сохранять инварианты (как в случае независимых компонентов x и y точки на плоскости) вполне допустимо предоставлять прямой доступ к публичным членам записи. С другой стороны, при наличии набора правил, определяющих, какова корректная форма записи, для ее обновления следует использовать приватные поля и методы, чтобы гарантировать соблюдение этих правил.

Еще один вариант: сделать поля класса неизменяемыми, как показано в листинге 3.7. В этом случае можно обеспечить корректное состояние записи при ее инициализации, а затем разрешить прямой доступ к членам класса, поскольку внешний код все равно не сможет их поменять.

Листинг 3.6. Класс Currency, сохраняющий инварианты

```
class Currency {
    private dollars: number = 0;
    private cents: number = 0;

    constructor(dollars: number, cents: number) {
        this.assignDollars(dollars);
        this.assignCents(cents);
    }

    getDollars(): number {
        return this.dollars;
    }

    assignDollars(dollars: number) {
        if (!Number.isSafeInteger(dollars) || dollars < 0)
            throw new Error();
        this.dollars = dollars;
    }

    getCents(): number {
        return this.cents;
    }

    assignCents(cents: number) {
        if (!Number.isSafeInteger(cents) || cents < 0)
            throw new Error();

        this.assignDollars(this.dollars + Math.floor(cents / 100));
        this.cents = cents % 100;
    }
}
```

Приватность членов dollars и cents гарантирует, что внешний код не сможет обойти проверку корректности значений

Если количество долларов или центов некорректно (отрицательное либо небезопасное целое число), то генерируется исключение

Если количество долларов или центов некорректно (отрицательное либо небезопасное целое число), то генерируется исключение

Нормализуем значение, преобразовывая сотни центов в доллары

Листинг 3.7. Класс Currency с неизменяемыми полями

```
class Currency {
    readonly dollars: number;
    readonly cents: number;

    constructor(dollars: number, cents: number) {
        if (!Number.isSafeInteger(cents) || cents < 0)
            throw new Error();

        dollars = dollars + Math.floor(cents / 100);
        cents = cents % 100;

        if (!Number.isSafeInteger(dollars) || dollars < 0)
            throw new Error();

        this.dollars = dollars;
        this.cents = cents;
    }
}
```

Поля dollars и cents публичные, но доступны только для чтения, после инициализации поменять их значения невозможно

Теперь вся проверка на корректность производится в конструкторе

В случае неизменяемых членов класса для соблюдения инвариантов больше не нужны функции их редактирования. Значения членов класса задаются только при инициализации, поэтому можно перенести в конструктор всю логику проверки на корректность. У неизменяемых данных есть и другие преимущества: гарантированная безопасность конкурентного обращения к таким данным из различных потоков выполнения, поскольку эти данные не меняются. Изменяемость данных может приводить к состоянию гонки, при котором один поток выполнения модифицирует используемое другим значение.

У записей с неизменяемыми членами есть недостаток: необходимо создавать новый экземпляр для каждого нового значения. В зависимости от затрат, требуемых для создания нового экземпляра, можно предпочесть использование либо записи, члены которой обновляются без создания дополнительных структур данных с помощью геттеров и сеттеров, либо реализации, в которой каждое обновление значений требует создания нового объекта.

Цель — предотвратить изменения со стороны внешнего кода, которые бы нарушали наши правила корректности значений. Это можно сделать с помощью приватных членов класса, перенаправляя все обращения к ним через методы, либо путем применения неизменяемых членов класса, проверяя на корректность в конструкторе.

3.1.4. Упражнение

Какое определение точки в 3D-пространстве предпочтительнее?

А. `type Point = [number, number, number];`.

Б. `type Point = number | number | number;`.

В. `type Point = { x: number, y: number, z: number };`.

Г. `type Point = any;`.

3.2. Выражаем строгую дизъюнкцию с помощью типов данных

До сих пор мы изучали сочетание типов путем такой их группировки, что значение включало по одному значению каждого из типов-компонентов. Другой основной способ сочетания типов — строгая дизъюнкция, при которой значение представляет собой любое одно из возможных наборов значений типов, лежащих в его основе (рис. 3.3).

3.2.1. Перечисляемые типы

Начнем с очень простой задачи: кодирования дня недели в системе типов. Можно кодировать день недели в виде числа от 0 до 6, где 0 — первый день недели, а 6 — последний. Это неидеальный вариант, поскольку у разных людей, работающих над кодом, представления о том, какой день недели считать первым, могут различаться. В таких странах, как США, Канада и Япония, первым днем

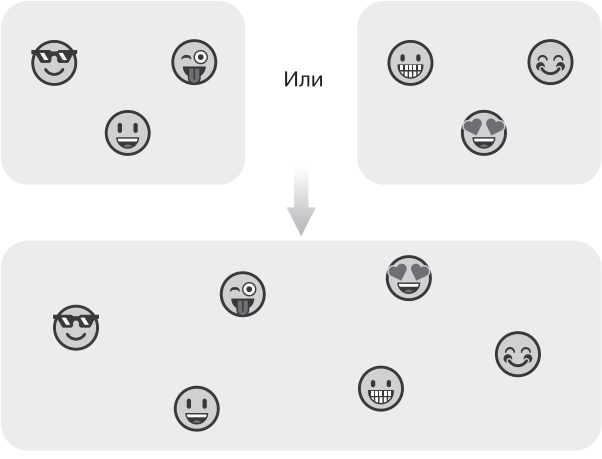


Рис. 3.3. Объединение двух типов таким образом, чтобы итоговый тип содержал значение одного из этих типов

недели считается воскресенье, а в стандарте ISO 8601 и большинстве европейских стран — понедельник (листинг 3.8).

Листинг 3.8. Кодирование дня недели с помощью числа

```
function isWeekend(dayOfWeek: number): boolean {
    return dayOfWeek == 5 || dayOfWeek == 6;
}

function isWeekday(dayOfWeek: number): boolean {
    return dayOfWeek >= 1 && dayOfWeek <= 5;
}
```

Европейский разработчик считает выходными дни 5 и 6 (субботу и воскресенье)

Американский разработчик считает будними дни с 1 по 5 (с понедельника по пятницу)

Из этого примера кода очевидно, что обе функции не могут быть правильными одновременно. Если `0` соответствует воскресенью, то функция `isWeekend` работает неправильно; если же `0` соответствует понедельнику, то некорректна функция `isWeekday`. К сожалению, автоматически предотвращать подобные ошибки невозможно, поскольку смысл значения `0` не обеспечивается системой типов, а определяется соглашением.

В качестве альтернативного варианта можно объявить набор констант, соответствующих дням недели, и использовать их везде, где ожидается день недели (листинг 3.9).

Эта реализация немного лучше предыдущей, но по-прежнему есть проблема: из объявления функций непонятно, какие значения ожидаются в качестве аргументов типа `number`. Как разработчику, незнакомому с кодом, догадаться по `dayOfWeek: number`, что необходимо использовать одну из приведенных констант? Он вообще может не знать, что эти константы определены где-то в каком-то модуле, и вместо них применять числа, как в нашем первом примере в листинге 3.8. Кроме того,

не исключено, что кто-нибудь вызовет такую функцию с совершенно недопустимыми аргументами, например `-1` или `10`. Лучшим решением будет объявить для дней недели перечисляемый тип данных (листинг 3.10).

Листинг 3.9. Кодирование дня недели с помощью констант

```
const Sunday: number = 0;
const Monday: number = 1;
const Tuesday: number = 2;
const Wednesday: number = 3;
const Thursday: number = 4;
const Friday: number = 5;
const Saturday: number = 6;

function isWeekend(dayOfWeek: number): boolean {
    return dayOfWeek == Saturday || dayOfWeek == Sunday;
}

function isWeekday(dayOfWeek: number): boolean {
    return dayOfWeek >= Monday && dayOfWeek <= Friday;
}
```

Вместо чисел мы теперь используем поименованные константы, гарантирующие согласованность кода

Листинг 3.10. Кодирование дня недели с помощью перечисляемого типа данных

```
enum DayOfWeek {
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
}

function isWeekend(dayOfWeek: DayOfWeek): boolean {
    return dayOfWeek == DayOfWeek.Saturday
        || dayOfWeek == DayOfWeek.Sunday;
}

function isWeekday(dayOfWeek: DayOfWeek): boolean {
    return dayOfWeek >= DayOfWeek.Monday
        && dayOfWeek <= DayOfWeek.Friday;
}
```

Заменяем константы на перечисляемый тип данных

Теперь у нас есть специальный тип данных, отражающий день недели

При таком подходе дни недели непосредственно кодируются в перечисляемом типе данных, что имеет два преимущества. Во-первых, отсутствует неоднозначность относительно того, какое число соответствует понедельнику и какое — воскресенью, поскольку все явно прописано в коде. Во-вторых, из объявления функции, ожидающей `dayOfWeek: DayOfWeek`, совершенно ясно, что необходимо передать в качестве аргумента не число, а член перечисления `DayOfWeek`.

Это простейший пример объединения набора значений в новый тип данных. Переменная этого типа может принимать одно из указанных значений. Перечисля-

емые типы данных имеет смысл использовать везде, где нужно однозначно представить маленькое множество вероятных значений. Посмотрим, как применить данную идею к типам вместо значений.

3.2.2. Опциональные типы данных

Допустим, нам нужно преобразовать переданное пользователем значение типа `string` в `DayOfWeek`. Если полученную строку можно интерпретировать как день недели, то необходимо вернуть значение `DayOfWeek`, в противном случае следует явным образом сообщить, что день недели — `undefined`. В TypeScript это можно реализовать с помощью оператора |, который позволяет сочетать типы, как показано в листинге 3.11.

Листинг 3.11. Разбор входных данных с преобразованием в DayOfWeek или undefined

```
function parseDayOfWeek(input: string): DayOfWeek | undefined {
    switch (input.toLowerCase()) {
        case "sunday": return DayOfWeek.Sunday;
        case "monday": return DayOfWeek.Monday;
        case "tuesday": return DayOfWeek.Tuesday;
        case "wednesday": return DayOfWeek.Wednesday;
        case "thursday": return DayOfWeek.Thursday;
        case "friday": return DayOfWeek.Friday;
        case "saturday": return DayOfWeek.Saturday;
        default: return undefined;
    }
}

function useInput(input: string) {
    let result: DayOfWeek | undefined = parseDayOfWeek(input);

    if (result === undefined) {
        console.log(`Failed to parse "${input}"`);
    }else {
        let dayOfWeek: DayOfWeek = result;
        /* используем DayOfWeek */
    }
}
```

Функция возвращает DayOfWeek или undefined

Если ни один из вариантов не подходит, то возвращаем undefined как сигнал того, что разобрать входные данные не удалось

Проверяем, удалось ли выполнить синтаксический разбор; если нет, то заносим в журнал сообщение об ошибке

Если результат не undefined, то извлекаем из него и используем значение типа DayOfWeek

Функция `parseDayOfWeek()` возвращает `DayOfWeek` или `undefined`, а функция `useInput` вызывает ее и пытается развернуть результат, занося в журнал сообщение об ошибке или получая пригодное для использования значение `DayOfWeek`.

ОПЦИОНАЛЬНЫЕ ТИПЫ ДАННЫХ

Опциональный тип данных (optional type), или просто опционал, отражает (вероятно, отсутствующее) значение другого типа T. Экземпляр опционального типа может содержать (любое) значение типа T или специальное значение, указывающее на отсутствие значения типа T.

Опционал своими руками

Некоторые из самых широко используемых языков программирования до сих пор не имеют синтаксической поддержки сочетания типов подобным образом, но основные конструкции доступны в виде библиотек. Наш пример с `DayOfWeek` или `undefined` представляет собой *опциональный тип*. Опционал содержит либо значение лежащего в его основе типа, либо индикатор отсутствия значения.

Опциональный тип данных обычно служит оберткой для другого типа, передаваемого в качестве обобщенного аргумента типа, и включает несколько методов: `hasValue()`, указывающий, содержит ли объект реальное значение, и `getValue()`, который возвращает значение. Попытка вызвать метод `getValue()`, когда значение отсутствует, приводит к генерации исключения, как показано в листинге 3.12.

Листинг 3.12. Опциональный тип данных

```
class Optional<T> {
    private value: T | undefined;
    private assigned: boolean;

    constructor(value?: T) {
        if (value) {
            this.value = value;
            this.assigned = true;
        }else {
            this.value = undefined;
            this.assigned = false;
        }
    }

    hasValue(): boolean {
        return this.assigned;
    }

    getValue(): T {
        if (!this.assigned) throw Error();
        return <T>this.value;
    }
}
```

Опционал служит оберткой для обобщенного типа данных T

value представляет собой необязательный аргумент, поскольку TypeScript не поддерживает перегрузку конструкторов

Если значение данного опционала не задано, то попытка получить его приводит к генерации исключения

В других языках, где нет оператора типа `|`, который позволяет задать тип `T | undefined`, можно воспользоваться типом, допускающим неопределенное значение. *Тип, допускающий неопределенное значение* (nullable type), может принимать любое значение типа либо значение `null`, отражающее отсутствие значения.

Возможно, вы удивляетесь, для чего может понадобиться опциональный тип данных, если в большинстве языков ссылочные типы данных могут принимать значение `null`, так что способ кодировать отсутствие значения уже есть и без подобного типа.

Отличие в том, что использование `null` может приводить к ошибкам (о чем сказано во врезке «Ошибка стоимостью миллиард долларов» ниже), поскольку непонятно, может ли конкретная переменная принимать значение `null`. Приходится добавлять проверки на `null` по всему коду или рисковать разыменованием переменной со зна-

чением null, приводящим к ошибке во время выполнения. Идея опционального типа данных состоит в расцеплении null с диапазоном допустимых значений. Всякий раз, встречая опционал, мы *знаем*, что он может не содержать никакого значения. И лишь после проверки, что в нем действительно содержится значение, мы можем «распаковать» опционал и получить переменную типа, лежащего в его основе. С этого момента мы точно *знаем*, что переменная не может быть null. Это различие отражается в системе типов: тип переменной (DayOfWeek | undefined или Optional<DayOfWeek>), которая «может быть null», отличается от «распакованного» значения, которое, как мы знаем, не может быть null (DayOfWeek). Несовместимость опционала и лежащего в его основе типа очень удобна, поскольку гарантирует невозможность случайно использовать опционал (в котором может не содержаться значения) вместо лежащего в его основе типа без распаковки значения явным образом.

Ошибка стоимостью миллиард долларов

Знаменитый специалист по теории вычислительной техники и лауреат премии Тьюринга Тони Хоар называет нулевые ссылки своей ошибкой стоимостью миллиард долларов. Цитируют следующее его высказывание: «Я называю изобретение нулевой ссылки в 1965 году своей ошибкой стоимостью миллиард долларов. В то время я проектировал первую комплексную систему типов для ссылок в объектно-ориентированном языке. Я ставил перед собой цель обеспечить абсолютную безопасность всех ссылок путем автоматической проверки их компилятором. Но я поддался искушению включить в язык нулевую ссылку просто потому, что ее было так легко реализовать. Это привело к бесчисленным ошибкам, уязвимостям и системным сбоям, причинившим, вероятно, за последние 40 лет неприятностей и убытков на миллиард долларов».

После десятилетий ошибок, связанных с разыменованием null, стало очевидно, что лучше будет не считать null (отсутствие значения) допустимым значением типа.

3.2.3. Результат или сообщение об ошибке

Расширим наш пример преобразования строки DayOfWeek так, чтобы при невозможности определить значение DayOfWeek возвращать не просто неопределенное значение, а более подробную информацию об ошибке. Желательно различать случай, когда строка пуста и мы не можем произвести ее разбор. Это удобно, если данный код используется для элемента управления текстовым вводом, для отображения пользователю различных сообщений в зависимости от ошибки (например, Please enter a day of week (Пожалуйста, введите день недели) или Invalid day of week (Недопустимый день недели)).

Часто встречающийся антипаттерн состоит в возврате *и* DayOfWeek, *и* кода ошибки, как показано в листинге 3.13. Если код ошибки указывает на успешное выполнение, то можно использовать значение DayOfWeek. Если же он указывает на ошибку, то значение DayOfWeek некорректно и его не следует применять.

Листинг 3.13. Возвращаем из функции результат и ошибку

```
enum InputError {
    OK,
    NoInput,
    Invalid
}

class Result {   #B
    error: InputError;
    value: DayOfWeek;

    constructor(error: InputError, value: DayOfWeek) {
        this.error = error;
        this.value = value;
    }
}

function parseDayOfWeek(input: string): Result {
    if (input == "")
        return new Result(InputError.NoInput, DayOfWeek.Sunday);

    switch (input.toLowerCase()) {
        case "sunday":
            return new Result(InputError.OK, DayOfWeek.Sunday);
        case "monday":
            return new Result(InputError.OK, DayOfWeek.Monday);
        case "tuesday":
            return new Result(InputError.OK, DayOfWeek.Tuesday);
        case "wednesday":
            return new Result(InputError.OK, DayOfWeek.Wednesday);
        case "thursday":
            return new Result(InputError.OK, DayOfWeek.Thursday);
        case "friday":
            return new Result(InputError.OK, DayOfWeek.Friday);
        case "saturday":
            return new Result(InputError.OK, DayOfWeek.Saturday);
        default:
            return new Result(InputError.Invalid, DayOfWeek.Sunday);
    }
}
```

InputError отражает код ошибки

Result сочетает код ошибки и значение DayOfWeek

Если строка пуста, то возвращаем NoInput и значение DayOfWeek по умолчанию

Возвращаем OK и результат разбора DayOfWeek, если этот разбор удалось выполнить

Если произвести разбор не удалось, то возвращаем Invalid и значение DayOfWeek по умолчанию

Этот вариант неидеален, ведь ничего не мешает нам задействовать `DayOfWeek`, даже если случайно забыть проверить код ошибки. Кроме того, может использоваться значение по умолчанию, и отнюдь не всегда мы различим, допустимо ли оно. Мы можем передать ошибку далее по системе, например записать ее в базу данных, не отдавая себе отчета, что это значение вообще не следовало применять.

Если посмотреть на это с точки зрения типов данных как множеств, то можно считать, что наш результат содержит комбинацию всех возможных кодов ошибок *и* всех возможных результатов (рис. 3.4).

Рис. 3.4. Все возможные значения типа Result как сочетание InputError и DayOfWeek. Всего 21 значение (3 InputError x 7 DayOfWeek)

Вместо этого желательно возвращать *либо* ошибку, *либо* допустимое значение. Если нам это удастся, то множество возможных значений резко сократится и будет исключен риск случайно использовать компонент `DayOfWeek` типа `Result`, в котором компонент `InputError` имеет значение `NoInput` или `Invalid` (рис. 3.5).

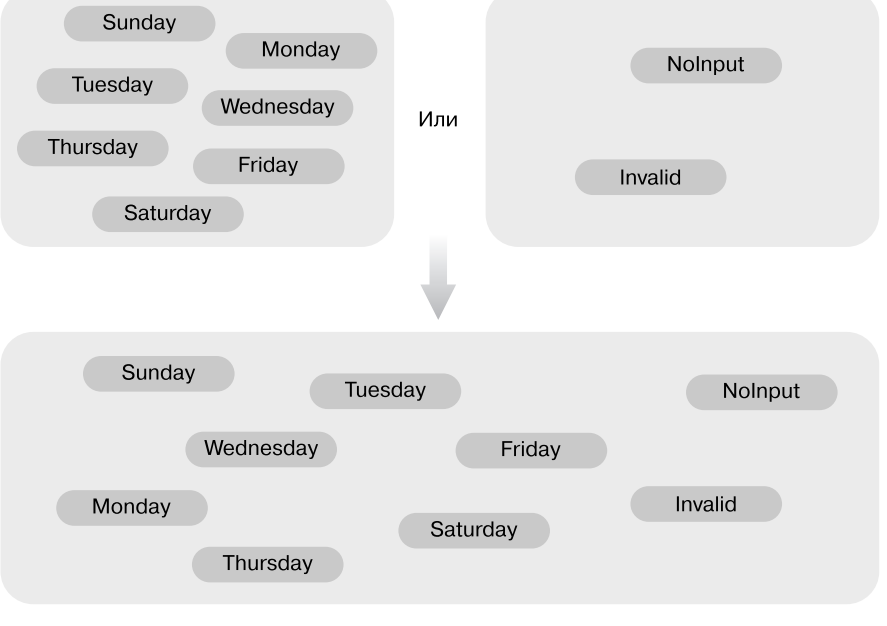


Рис. 3.5. Все возможные значения типа Result как сочетание InputError или DayOfWeek. Всего девять значений (2 InputError + 7 DayOfWeek). Больше не требуется InputError со значением OK, поскольку само наличие значения DayOfWeek указывает на отсутствие ошибки

XOR-тип данных своими руками

Наш XOR-тип данных `Either` будет оберткой для двух типов, `TLeft` и `TRight`, где `TLeft` станет служить для хранения типа ошибки, а `TRight` — типа допустимого значения (если ошибки нет, то значение «правильное»[1]) (листинг 3.14). Напомню, в некоторых языках программирования это включено в стандартную библиотеку, но при необходимости можно легко реализовать подобный тип данных.

Листинг 3.14. Тип Either

```
class Either<TLeft, TRight> {
    private readonly value: TLeft | TRight;
    private readonly left: boolean;

    private constructor(value: TLeft | TRight, left: boolean) {
        this.value = value;
        this.left = left;
    }

    isLeft(): boolean {
        return this.left;
    }

    getLeft(): TLeft {
        if (!this.isLeft()) throw new Error();

        return <TLeft>this.value;
    }

    isRight(): boolean {
        return !this.left;
    }

    getRight(): TRight {
        if (!this.isRight()) throw new Error();

        return <TRight>this.value;
    }

    static makeLeft<TLeft, TRight>(value: TLeft) {
        return new Either<TLeft, TRight>(value, true);
    }

    static makeRight<TLeft, TRight>(value: TRight) {
        return new Either<TLeft, TRight>(value, false);
    }
}
```

Данный тип служит оберткой для значения типа TLeft или Tright, а также флага, указывающего, какой именно тип используется

Конструктор приватный, поскольку нам нужна уверенность в согласованности значения и булева флага

Попытка получить TLeft при наличии TRight или наоборот приводит к генерации ошибки

Функции-фабрики вызывают конструктор и обеспечивают согласованность булева флага со значением

[1] В оригинале игра слов: по-английски «правильный» и «правый» (right) — омонимы. — *Примеч. пер.*

В языках, где отсутствует оператор типа |, можно просто использовать значение общего типа, например `Object` в Java и C#. Преобразование обратно в типы `TLeft` и `TRight` осуществляется методами `getLeft()` и `getRight()` соответственно.

С помощью подобного типа можно усовершенствовать нашу реализацию `parseDayOfWeek()` так, чтобы она возвращала результат типа `Either<InputError, DayOfWeek>`, делая невозможным дальнейшее распространение по системе некорректного или используемого по умолчанию значения `DayOfWeek`. Если функция возвращает `InputError`, то в результате отсутствует `DayOfWeek` и попытка извлечь его с помощью вызова `getLeft()` приводит к генерации ошибки (листинг 3.15).

Нам опять приходится явным образом распаковывать значение. Если мы знаем точно, что значение допустимое (`isLeft()` возвращает `true`), и извлекаем его с помощью метода `getLeft()`, то гарантированно получим корректные данные.

Листинг 3.15. Возвращаем из функции результат или ошибку

```
enum InputError {
    NoInput,
    Invalid
}

type Result = Either<InputError, DayOfWeek>;

function parseDayOfWeek(input: string): Result {
    if (input == "")
        return Either.makeLeft(InputError.NoInput);

    switch (input.toLowerCase()) {
        case "sunday":
            return Either.makeRight(DayOfWeek.Sunday);
        case "monday":
            return Either.makeRight(DayOfWeek.Monday);
        case "tuesday":
            return Either.makeRight(DayOfWeek.Tuesday);
        case "wednesday":
            return Either.makeRight(DayOfWeek.Wednesday);
        case "thursday":
            return Either.makeRight(DayOfWeek.Thursday);
        case "friday":
            return Either.makeRight(DayOfWeek.Friday);
        case "saturday":
            return Either.makeRight(DayOfWeek.Saturday);
        default:
            return Either.makeLeft(InputError.Invalid);
    }
}
```

Больше не требуется InputError со значением OK. Мы получаем либо ошибку, либо значение

Result теперь представляет собой либо InputError, либо DayOfWeek вместо сочетания того и другого

Возвращаем результат или ошибку с помощью методов Either.makeRight и Either.makeLeft

Возвращаем результат или ошибку с помощью методов Either.makeRight и Either.makeLeft

Эта усовершенствованная реализация использует систему типов для исключения некорректных состояний, каких как (`NoInput, Sunday`), из которых мы могли случайно применить значение `Sunday`. Кроме того, не требуется `InputError` со значением `OK`, поскольку при успешном разборе сообщение об ошибке отсутствует.

Исключения

Генерация исключения при ошибке — прекрасный пример того, как возвращается результат или ошибка: функция либо возвращает результат, либо генерирует исключение. В некоторых случаях использовать исключения нельзя и лучше задействовать тип `Either`. Это пригодится в следующих ситуациях: для распространения ошибки по процессам или потокам выполнения; в качестве принципа проектирования, когда сама ошибка не носит характера исключения (частый случай при обработке вводимых пользователем данных); при вызове API операционной системы, использующих коды ошибок, и т. д. В подобных случаях, когда генерация исключения невозможна или нежелательна, но необходимо сообщить о наличии значения или неудачном разборе, лучше всего кодировать подобное сообщение в виде *«значение или ошибка»*, а не *«значение и ошибка»*.

Если же генерация исключений приемлема, то можно использовать их для дополнительной уверенности, что мы не получим в итоге некорректный результат *и* ошибку. При генерации исключения функция больше не выполняет «обычный» возврат, передавая значение вызывающей стороне с помощью оператора `return`. Вместо этого объект исключения проходит по системе до тех пор, пока не встретится соответствующий оператор `catch`. Таким образом, мы получаем или результат, или исключение. Мы не станем описывать подробно генерацию исключений, поскольку, несмотря на то что во многих языках программирования есть возможности генерации и перехвата исключений, типы исключения не выделяются ничем особенным.

3.2.4. Вариантные типы данных

Мы рассмотрели опциональные типы данных, которые содержат значение определенного типа либо не содержат никакого значения. Затем изучили XOR-типы данных, содержащие значение либо `TLeft`, либо `TRight`. Обобщением их являются *вариантные типы данных* (variant types).

ВАРИАНТНЫЕ ТИПЫ ДАННЫХ

> Вариантные типы данных, известные также как маркированные объединения (tagged unions), могут содержать значение любого из нескольких типов, лежащих в их основе. Маркированные потому, что, даже если диапазоны значений таких типов пересекаются, мы все равно сможем точно сказать, к какому из них относится конкретное значение.

Рассмотрим в листинге 3.16 пример с набором геометрических фигур. У каждой из них свой набор свойств и метка (реализованная в виде свойства `kind`). Опишем тип, представляющий собой объединение типов всех фигур. Далее при необходимости, например, визуализировать эти фигуры можно воспользоваться их свойствами

kind для определения того, какой фигурой является конкретный экземпляр, и привести его к соответствующему типу фигуры. Этот процесс аналогичен распаковке в предыдущих примерах.

Листинг 3.16. Маркированное объединение фигур

```
class Point {
    readonly kind: string = "Point";
    x: number = 0;
    y: number = 0;
}

class Circle {
    readonly kind: string = "Circle";
    x: number = 0;
    y: number = 0;
    radius: number = 0;
}

class Rectangle {
    readonly kind: string = "Rectangle";
    x: number = 0;
    y: number = 0;
    width: number = 0;
    height: number = 0;
}

type Shape = Point | Circle | Rectangle;

let shapes: Shape[] = [new Circle(), new Rectangle()];

for (let shape of shapes) {
    switch (shape.kind) {
        case "Point":
            let point: Point = <Point>shape;
            console.log(`Point ${JSON.stringify(point)}`);
            break;
        case "Circle":
            let circle: Circle = <Circle>shape;
            console.log(`Circle ${JSON.stringify(circle)}`);
            break;
        case "Rectangle":
            let rectangle: Rectangle = <Rectangle>shape;
            console.log(`Rectangle ${JSON.stringify(rectangle)}`);
            break;
        default:
            throw new Error();
    }
}
```

Проходим в цикле по фигурам и проверяем свойство kind каждой из них

Если значение свойства kind равно "Point" (Точка), то можно безопасно приводить к типу Point. То же самое справедливо относительно "Circle" (Круг) и "Rectangle" (Прямоугольник)

Если разновидность фигуры неизвестна, то генерируем ошибку. Это значит, что неким образом в объединение попал какой-то другой тип, чего быть не должно

В предыдущем примере член kind во всех классах отражает метку, указывающую фактический тип значения. Значение поля shape.kind указывает, является ли экземпляр Shape на самом деле Point, Circle или Rectangle. Можно также реализовать

обобщенный вариантный тип данных, который самостоятельно отслеживает тип данных и не требует хранения метки в самих типах.

Реализуем простой вариантный тип данных, способный хранить значение одного из трех типов данных и отслеживать фактически хранимый тип на основе индекса типа.

Вариантный тип данных своими руками

Различные языки программирования предоставляют разные возможности обобщения и проверки типов. Так, одни языки допускают переменное количество обобщенных аргументов (поэтому вариантные типы данных могут основываться на любом числе типов). Другие дают различные способы определения, к какому типу относится значение, на этапе как компиляции, так и выполнения.

У следующей реализации на TypeScript есть свои достоинства и недостатки, которые, возможно, отличаются от других языков программирования (листинг 3.17). Она является отправным пунктом для создания обобщенного вариантного типа данных. Однако, скажем, в Java или C# ее нужно реализовывать иначе. TypeScript, например, не поддерживает перегрузки методов, а в других языках можно обойтись одной функцией `make()`, перегруженной для каждого из обобщенных типов.

Листинг 3.17. Тип данных Variant

```
class Variant<T1, T2, T3> {
    readonly value: T1 | T2 | T3;
    readonly index: number;

    private constructor(value: T1 | T2 | T3, index: number) {
        this.value = value;
        this.index = index;
    }

    static make1<T1, T2, T3>(value: T1): Variant<T1, T2, T3> {
        return new Variant<T1, T2, T3>(value, 0);
    }

    static make2<T1, T2, T3>(value: T2): Variant<T1, T2, T3> {
        return new Variant<T1, T2, T3>(value, 1);
    }

    static make3<T1, T2, T3>(value: T3): Variant<T1, T2, T3> {
        return new Variant<T1, T2, T3>(value, 2);
    }
}
```

Эта реализация берет на себя хранение меток, так что теперь можно исключить их из типов геометрических фигур (листинг 3.18).

Может показаться, что эта реализация не имеет особых преимуществ; мы пришли к использованию числовых меток и произвольно выбрали метку `0` для `Point` и `1` —

для `Circle`. Возможно, вы недоумеваете, почему мы вместо этого не применили для наших фигур иерархию классов с базовым методом, который реализовывал бы каждый тип.

Листинг 3.18. Объединение геометрических фигур как вариантный тип данных

```
class Point {
    x: number = 0;
    y: number = 0;
}

class Circle {
    x: number = 0;
    y: number = 0;
    radius: number = 0;
}

class Rectangle {
    x: number = 0;
    y: number = 0;
    width: number = 0;
    height: number = 0;
}

type Shape = Variant<Point, Circle, Rectangle>;

let shapes: Shape[] = [
    Variant.make2(new Circle()),
    Variant.make3(new Rectangle())
];

for (let shape of shapes) {
    switch (shape.index) {
        case 0:
            let point: Point = <Point>shape.value;
            console.log(`Point ${JSON.stringify(point)}`);
            break;
        case 1:
            let circle: Circle = <Circle>shape.value;
            console.log(`Circle ${JSON.stringify(circle)}`);
            break;
        case 2:
            let rectangle: Rectangle = <Rectangle>shape.value;
            console.log(`Rectangle ${JSON.stringify(rectangle)}`);
            break;
        default:
            throw new Error();
    }
}
```

Больше не нужно хранить метки в самих фигурах

Класс Shape теперь представляет собой Variant на основе этих трех типов данных

Чтобы выяснить метку, мы анализируем свойство index, а для получения самого объекта используем свойство value

Для решения этой задачи нам понадобится изучить паттерн проектирования «Посетитель» и способы его реализации.

3.2.5. Упражнения

1. Пользователи передают значение для выбора из нескольких цветов (красный, зеленый и синий). Каким должен быть тип этого значения?

   А. `number` с `Red = 0, Green = 1, Blue = 2`.

   Б. `string` с `Red = "Red", Green = "Green", Blue = "Blue"`.

   В. `enum Colors { Red, Green, Blue }`.

   Г. `type Colors = Red | Green | Blue`, где цвета являются классами.

2. Значение какого типа должна возвращать функция, получающая на входе строку и производящая разбор этой строки в числовое значение? Функция не генерирует исключений.

   А. `number`.

   Б. `number | undefined`.

   В. `Optional<number>`.

   Г. Или Б, или В.

3. В операционных системах коды ошибок обычно представляют собой числовые значения. Каков должен быть возвращаемый тип функции, которая может возвращать либо числовое значение, либо числовой код ошибки?

   А. `number`.

   Б. `{ value: number, error: number }`.

   В. `number | number`.

   Г. `Either<number, number>`.

3.3. Паттерн проектирования «Посетитель»

Обсудим паттерн «Посетитель» и обход элементов документа — сначала с точки зрения объектно-ориентированного программирования, а затем с помощью реализованного нами обобщенного типа маркированного объединения. Если вы не знакомы с этим паттерном, то не волнуйтесь, я расскажу, что он собой представляет, по мере работы над примером.

Мы начнем с «наивной» реализации, увидим, как паттерн позволяет усовершенствовать архитектуру, а затем рассмотрим альтернативную реализацию, в которой не требуются иерархии классов.

Начнем с трех элементов документа: с абзаца (paragraph), изображения (picture) и таблицы (table). Нам нужно либо визуализировать их на экране, либо обеспечить их прочтение вслух утилитой чтения с экрана для слабовидящих пользователей.

3.3.1. «Наивная» реализация

Один из возможных подходов — создание общего интерфейса, чтобы все элементы умели визуализироваться на экране или обеспечивать чтение вслух, как показано в листинге 3.19.

Листинг 3.19. «Наивная» реализация

Эти два класса предоставляют методы для визуализации и чтения вслух, для краткости опущенные в этом листинге

```
class Renderer {/* методы для визуализации*/ }
class ScreenReader {/* методы для чтения содержимого экрана */ }

interface IDocumentItem {
    render(renderer: Renderer): void;
    read(screenReader: ScreenReader): void;
}

class Paragraph implements IDocumentItem {
 /* члены класса Paragraph опущены*/
    render(renderer: Renderer) {
 /* использует renderer для своего отображения на экране */
    }

    read(screenReader: ScreenReader) {
 /* использует screenReader для чтения себя вслух*/
    }
}

class Picture implements IDocumentItem {
    /* члены класса Picture опущены */
    render(renderer: Renderer) {
 /* использует renderer для своего отображения на экране */
    }

    read(screenReader: ScreenReader) {
        /* использует screenReader для чтения себя вслух*/
    }
}

class Table implements IDocumentItem {
    /* члены класса Table опущены */
    render(renderer: Renderer) {
        /* использует renderer для своего отображения на экране */
    }

    read(screenReader: ScreenReader) {
        /* использует screenReader для чтения себя вслух*/
    }
}

let doc: IDocumentItem[] = [new Paragraph(), new Table()];
let renderer: Renderer = new Renderer();

for (let item of doc) {
    item.render(renderer);
}
```

Интерфейс IDocumentItem определяет, что каждый элемент может визуализировать себя и обеспечить прочтение себя вслух

Элементы документа реализуют интерфейс IDocumentItem и с помощью средства визуализации или средства чтения содержимого экрана визуализируют себя либо читают себя вслух

С точки зрения архитектуры это не лучший подход. В элементах документа должна храниться информация, описывающая его содержимое, например текст или изображения, они не должны отвечать за все остальное, скажем за визуализацию и доступ.

Наличие кода для визуализации и обеспечения доступа в каждом из элементов документа сильно раздувает код. Хуже того, если нужно добавить новую возможность (например, возможность печати), то для ее реализации придется модифицировать описание интерфейса и всех реализующих его классов.

3.3.2. Использование паттерна «Посетитель»

Данный паттерн описывает операцию, которую необходимо выполнить над элементами объектной структуры данных. Он позволяет описать новую операцию без изменения классов элементов, над которыми она производится.

В нашем примере, приведенном в листинге 3.20, данный паттерн позволяет добавить новую возможность, вообще не трогая код элементов документа. Реализовать это можно с помощью *механизма двойной диспетчеризации* (double-dispatch mechanism), при котором элементы документа получают в качестве параметра объект-посетитель, после чего передают себя самих в него. Посетитель знает, как следует обрабатывать каждый из элементов (визуализировать, прочитать вслух и т. д.), и выполняет нужную операцию над переданным ему экземпляром элемента (рис. 3.6).

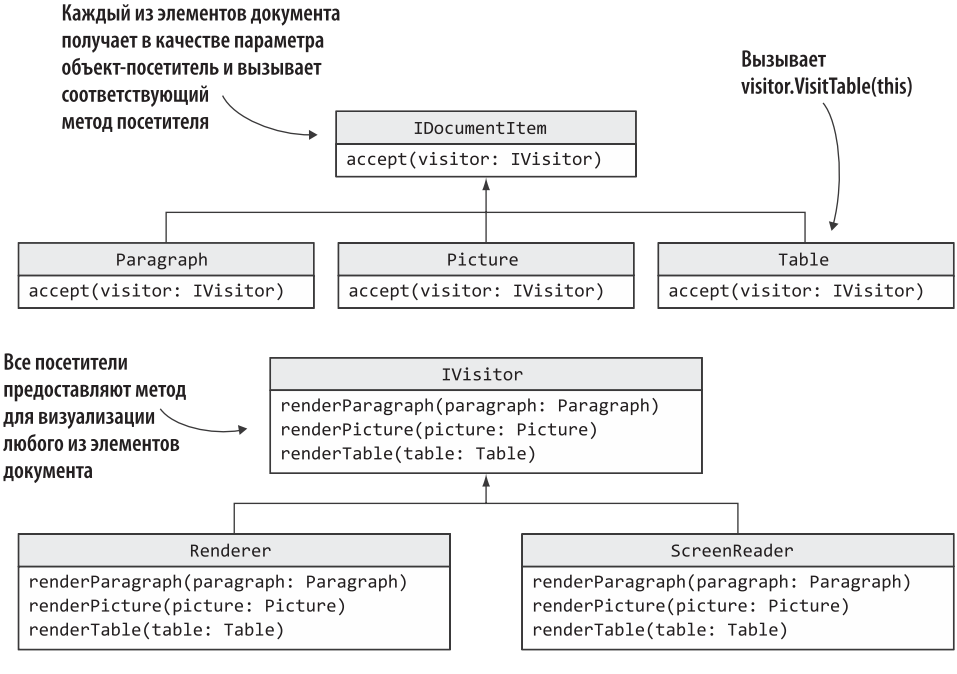


Рис. 3.6. Паттерн «Посетитель». Благодаря интерфейсу IDocumentItem у каждого из элементов документа есть метод accept(), принимающий в качестве аргумента экземпляр IVisitor. Он обеспечивает возможность обработки любым объектом-посетителем всех возможных типов элементов документа. Все элементы реализуют метод accept() для передачи себя посетителю. Благодаря этому паттерну можно разделять обязанности, такие как визуализация на экране и обеспечение доступа, между отдельными компонентами (посетителями), абстрагируя их от элементов документа

Листинг 3.20. Обработка с помощью паттерна «Посетитель»

```
interface IVisitor {
    visitParagraph(paragraph: Paragraph): void;
    visitPicture(picture: Picture): void;
    visitTable(table: Table): void;
}

class Renderer implements IVisitor {
    visitParagraph(paragraph: Paragraph) {/* ... */ }
    visitPicture(picture: Picture) {/* ... */ }
    visitTable(table: Table) {/* ... */ }
}

class ScreenReader implements IVisitor {
    visitParagraph(paragraph: Paragraph) {/* ... */ }
    visitPicture(picture: Picture) {/* ... */ }
    visitTable(table: Table) {/* ... */ }
}

interface IDocumentItem {
    accept(visitor: IVisitor): void;
}

class Paragraph implements IDocumentItem {
    /* члены класса Paragraph опущены */
    accept(visitor: IVisitor) {
        visitor.visitParagraph(this);
    }
}

class Picture implements IDocumentItem {
    /* члены класса Picture опущены */
    accept(visitor: IVisitor) {
        visitor.visitPicture(this);
    }
}

class Table implements IDocumentItem {
    /* члены класса Table опущены */
    accept(visitor: IVisitor) {
        visitor.visitTable(this);
    }
}

let doc: IDocumentItem[] = [new Paragraph(), new Table()];
let renderer: IVisitor = new Renderer();

for (let item of doc) {
    item.accept(renderer);
}
```

Интерфейс IVisitor обеспечивает возможность обработки любой из фигур каждым из посетителей

Этот интерфейс реализуют конкретные классы Renderer и ScreenReader

Теперь элементы документа должны всего лишь реализовать метод accept(), принимающий в качестве аргумента любой посетитель

Элементы вызывают соответствующий метод посетителя и передают себя в качестве аргументов

Термин *«двойная диспетчеризация»* обязан своим названием тому факту, что благодаря интерфейсу `IDocumentItem` сначала вызывается соответствующий метод

`accept()`; а затем в соответствии с полученным аргументом `IVisitor` вызывается соответствующая операция.

Теперь посетитель может пройти по набору объектов `IDocumentItem`, обрабатывая их с помощью вызова метода `accept()` для каждого из них. Ответственность за обработку теперь лежит на посетителях вместо самих элементов. Добавление нового посетителя никак не влияет на элементы документа. Новый посетитель должен лишь реализовать интерфейс `IVisitor`, и элементы документа примут его так же, как и любой другой.

Новый класс-посетитель `Printer` мог бы, например, реализовывать логику для распечатки абзаца, изображения и таблицы в методах `visitParagraph()`, `visitPicture()` и `visitTable()`. И элементам документа не понадобятся никакие изменения, чтобы стать доступными для печати.

Данный пример — классическая реализация паттерна «Посетитель». Теперь посмотрим, как выполнить нечто похожее с помощью вариантного типа данных.

3.3.3. Посетитель-вариант

Для начала снова обратимся к нашему обобщенному вариантному типу данных и реализуем функцию `visit()`. Она принимает в качестве аргументов объект вариантного типа данных и набор функций по одной для каждого типа и (в зависимости от значения, имеющегося в варианте) применяет к нему соответствующую функцию (листинг 3.21).

Листинг 3.21. Посетитель-вариант

```
function visit<T1, T2, T3>(
    variant: Variant<T1, T2, T3>,
    func1: (value: T1) => void,
    func2: (value: T2) => void,
    func3: (value: T3) => void
): void {
    switch (variant.index) {
        case 0: func1(<T1>variant.value); break;
        case 1: func2(<T2>variant.value); break;
        case 2: func3(<T3>variant.value); break;
        default: throw new Error();
    }
}
```

Функция visit() принимает в качестве аргументов по функции для каждого типа, из которых состоит наш вариантный тип данных

В зависимости от значения index вызывается функция, соответствующая типу значения, хранимого в варианте

Поместив элементы документа в вариантный тип данных, можно будет воспользоваться этой функцией для выбора соответствующего метода посетителя. При этом никаким из наших классов больше не нужно реализовывать определенные интерфейсы: ответственность за подбор нужного метода-обработчика для элемента документа возлагается теперь на обобщенную функцию `visit()`.

При таком подходе механизм двойной диспетчеризации расцепляется с используемыми нами типами и переносится в посетитель-вариант. Оба они представляют собой обобщенные типы данных, которые можно использовать повторно для предметных областей различных задач (листинг 3.22). Преимущество этого подхода

таково: посетители отвечают лишь за обработку, а элементы документа — только за хранение данных предметной области (рис. 3.7).

Листинг 3.22. Альтернативный способ обработки с помощью посетителя-варианта

```
class Renderer {
    renderParagraph(paragraph: Paragraph) {/* ... */ }
    renderPicture(picture: Picture) {/* ... */ }
    renderTable(table: Table) {/* ... */ }
}

class ScreenReader {
    readParagraph(paragraph: Paragraph) {/* ... */ }
    readPicture(picture: Picture) {/* ... */ }
    readTable(table: Table) {/* ... */ }
}

class Paragraph {
    /* члены класса Paragraph опущены */
}

class Picture {
    /* члены класса Picture опущены */
}

class Table {
    /* члены класса Table опущены */
}

let doc: Variant<Paragraph, Picture, Table>[] = [
    Variant.make1(new Paragraph()),
    Variant.make3(new Table())
];

let renderer: Renderer = new Renderer();

for (let item of doc) {
    visit(item,
        (paragraph: Paragraph) => renderer.renderParagraph(paragraph),
        (picture: Picture) => renderer.renderPicture(picture),
        (table: Table) => renderer.renderTable(table)
    );
}
```

Общий интерфейс для элементов документа больше не нужен

Элементы документа хранятся в вариантном типе данных, подходящем для хранения любого из возможных элементов

Функция visit подбирает для элемента соответствующий метод обработки

Кроме того, предполагается, что вариантный тип данных будет использоваться с помощью созданной нами функции `visit()`. Выяснение, какой именно тип содержит вариантный тип, на основе поля `index` чревато ошибками. Но обычно мы не извлекаем значение из этого типа, а применяем к нему различные функции с помощью метода `visit()`. Таким образом, чреватый ошибками выбор производится в реализации метода `visit()`, и мы можем об этом не думать. Инкапсуляция небезопасного в смысле ошибок кода в повторно используемом компоненте — рекомендуемая практика для снижения риска, поскольку позволяет положиться во множестве сценариев использования на одну проверенную реализацию.

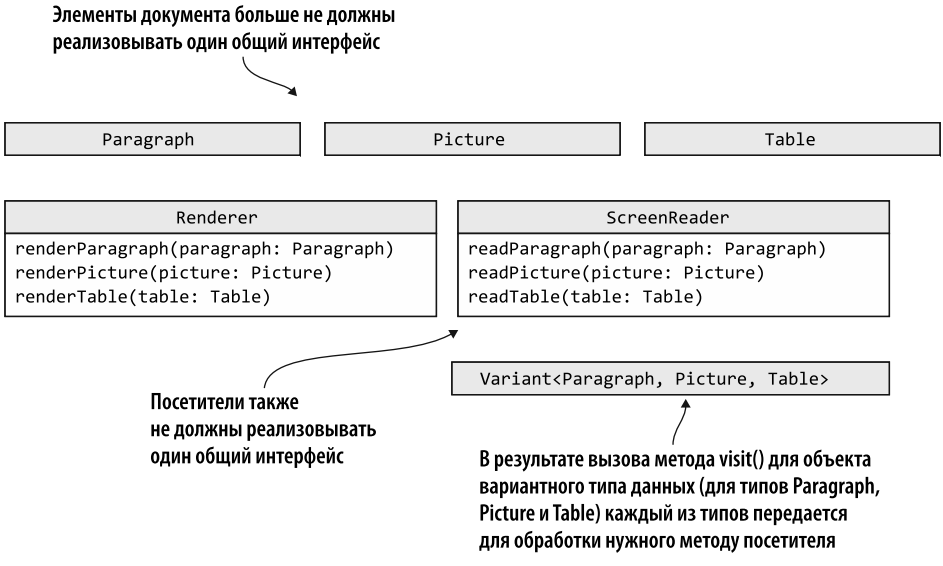


Рис. 3.7. Упрощенный паттерн «Посетитель»: теперь от элементов документа и посетителей не требуется реализация никаких интерфейсов. Сравните с рис. 3.6. Ответственность за подбор нужного метода-обработчика для элемента документа инкапсулируется в методе visit(). Как видно из рисунка, типы не связаны, что хорошо, поскольку программа становится более гибкой

Преимущество применения посетителя на основе вариантного типа данных вместо традиционной объектно-ориентированной реализации — полное разделение объектов предметной области с посетителями. Нам больше не требуется даже метод `accept()`, а элементы документа могут вообще ничего не знать о коде, который будет обрабатывать их. Не должны они и соответствовать какому-либо конкретному интерфейсу, такому как `IDocumentItem` в нашем примере. А все потому, что в типе `Variant` и его функции `visit()` инкапсулирован связующий код, который подбирает для фигур соответствующий посетитель.

3.3.4. Упражнение

Наша реализация метода `visit()` возвращает `void`. Расширьте ее так, чтобы она, получая на входе `Variant<T1, T2, T3>`, возвращала `Variant<U1, U2, U3>` с помощью одной из трех функций: `(value: T1) => U1`, или `(value: T2) => U2`, или `(value: T3) => U3`.

3.4. Алгебраические типы данных

Возможно, вы уже слышали термин *«алгебраические типы данных»* (algebraic data types, ADT). Они представляют собой способы сочетания типов в системе типов. На самом деле именно их мы и обсуждали на протяжении всей этой главы. ADT позволяют сочетать типы двумя способами, создавая типы-произведения и типы-суммы.

3.4.1. Типы-произведения

Типы-произведения (product types) — то, что мы называли в этой главе *составными типами данных* (compound types). Кортежи и записи относятся к типам-произведениям, поскольку их значения являются декартовыми произведениями составляющих их типов. Объединение типа `A = {a1, a2}` (тип `A` с возможными значениями `a1` и `a2`) и типа `B = {b1, b2}` (тип `B` с возможными значениями `b1` и `b2`) представляет собой тип-кортеж `<A, B>` вида `A x B = {(a1, b1), (a1, b2), (a2, b1), (a2, b2)}`.

ТИПЫ-ПРОИЗВЕДЕНИЯ

Типы-произведения объединяют несколько типов в новый тип, в котором хранится по значению каждого из этих типов. Тип-произведение типов A, B и C — который можно записать в виде A x B x C — содержит значение типа A, значение типа B и значение типа C. Примерами типов-произведений могут служить кортежи и записи. Кроме того, записи позволяют присваивать их компонентам осмысленные названия.

Типы-записи должны быть вам хорошо знакомы, ведь это обычно первый метод сочетания типов, изучаемый программистами-новичками. В последнее время в наиболее широко востребованных языках программирования начали применяться кортежи, но разобраться с ними совсем не сложно. Они очень похожи на записи. Отличие состоит в том, что нельзя дать названия их членам и их обычно можно описывать аналогично встроенным (inline) функциям/выражениям, указывая составляющие кортеж типы. В TypeScript, например, тип-кортеж, состоящий из двух значений типа `number`, можно описать в виде `[number, number]`.

Сначала мы обсудили типы-произведения, поскольку они должны быть более привычными для вас. Практически во всех языках программирования есть способы описания типов записей. А вот синтаксическая поддержка типов-сумм присутствует в меньшем числе широко используемых языков программирования.

3.4.2. Типы-суммы

Типы-суммы (sum types) — то, что мы называли в этой главе *XOR-типами данных* (either-or types). Они объединяют несколько типов в один, который может содержать значение любого из типов-компонентов, но только одного из них. Объединение типа `A = {a1, a2}` и типа `B = {b1, b2}` представляет собой тип-сумму `A|B` вида `A + B = {a1, a2, b1, b2}`.

ТИПЫ-СУММЫ

Типы-суммы объединяют несколько типов в новый тип, в котором хранится значение одного любого из этих типов. Тип-сумма типов A, B и C — который можно записать в виде A + B + C — содержит значение типа A, или значение типа B, или значение типа C. Примерами типов-сумм могут служить опционалы и вариантные типы данных.

Как мы видели, в языке TypeScript есть оператор типа `|`, но часто встречающиеся типы-суммы, например `Optional`, `Either` и `Variant`, реализуются и без его помощи. Эти типы обеспечивают широкие возможности представления результата или ошибки и закрытых множеств типов, предоставляя различные способы реализации распространенного паттерна «Посетитель».

В общем случае типы-суммы позволяют хранить значения не связанных друг с другом типов в одной переменной. Как показано в примере с паттерном «Посетитель», в качестве объектно-ориентированной альтернативы типам-суммам можно использовать общий базовый класс или интерфейс, но такое решение плохо масштабируется. Сочетание и комбинирование разных типов в различных местах приложения неизбежно приведет к огромному количеству интерфейсов или базовых классов, которые не слишком пригодны для повторного использования. Типы-суммы — это простой и аккуратный способ сочетания типов для подобных сценариев.

3.4.3. Упражнения

1. Какую разновидность типа описывает следующий оператор?

   ```
   let x: [number, string] = [42, "Hello"];
   ```

   А. Простой тип данных.

   Б. Тип-сумму.

   В. Тип-произведение.

   Г. И тип-сумму, и тип-произведение.

2. Какую разновидность типа описывает следующий оператор?

   ```
   let y: number | string = "Hello";
   ```

   А. Простой тип данных.

   Б. Тип-сумму.

   В. Тип-произведение.

   Г. И тип-сумму, и тип-произведение.

3. Допустим, заданы типы `enum Two { A, B }` и `enum Three { C, D, E }`. Каково количество возможных значений типа-кортежа `[Two, Three]`?

   А. 2.

   Б. 5.

   В. 6.

   Г. 8.

4. Допустим, заданы типы `enum Two { A, B }` и `enum Three { C, D, E }`. Каково количество возможных значений типа `Two | Three`?

   А. 2.

   Б. 5.

   В. 6.

   Г. 8.

Резюме

- Типы-произведения — это кортежи и записи, группирующие значения из нескольких типов.
- Записи позволяют задать названия членов записи, то есть придать им определенный смысл. Кроме того, оставляют меньше возможностей для путаницы, чем кортежи.
- Инварианты — правила, которым должна соответствовать корректно заданная запись. Обеспечить соблюдение инвариантов обладающего ими типа и невозможность их нарушения внешним кодом можно, объявив члены этого типа как `private` или `readonly`.
- Типы-суммы группируют типы по принципу «или-или» и содержат значения только одного из типов-компонентов.
- Функция должна возвращать значение *или* ошибку, а не значение *и* ошибку.
- Опциональный тип данных может содержать значение лежащего в его основе типа либо не содержать ничего. Риск ошибок снижается, если отсутствие значения не входит в состав области значений переменной (ошибка стоимостью миллиард долларов с указателем `null`).
- XOR-типы данных содержат значение левого или правого типа. По традиции правый тип соответствует корректному значению, а левый — ошибке.
- Вариантный тип данных может содержать значение из любого числа лежащих в его основе типов и позволяет выражать значения закрытых множеств типов, никак не связанных между собой (без каких-либо общих интерфейсов или базовых типов).
- Функция-посетитель, служащая для применения нужной функции к объекту вариантного типа данных, позволяет реализовать паттерн проектирования «Посетитель» другим способом, обеспечивающим лучшее разделение обязанностей.

В этой главе мы рассмотрели разнообразные способы создания новых типов данных путем сочетания уже существующих. В главе 4 мы увидим, как можно повысить безопасность программы благодаря кодированию смыслов с помощью системы типов и ограничения диапазонов допустимых значений типов. Кроме того, мы научимся добавлять и убирать информацию о типе и применять это к таким сценариям, как сериализация.

Ответы к упражнениям

3.1. Составные типы данных

В — оптимальным подходом будет задать названия для трех компонентов координат.

3.2. Выражаем строгую дизъюнкцию с помощью типов данных

1. В — в данном случае уместен перечисляемый тип данных. При подобных требованиях классы не нужны.
2. Г — допустимым возвращаемым типом в данном случае будет или встроенный тип-сумма, или `Optional`, поскольку и тот и другой могут выражать отсутствие значения.
3. Г — лучше всего использовать тип-маркированное объединение (`number | number` не позволит отличить, отражает ли данное значение ошибку).

3.3. Паттерн проектирования «Посетитель»

Вот одна из возможных реализаций:

```
function visit<T1, T2, T3, U1, U2, U3>(
    variant: Variant<T1, T2, T3>,
    func1: (value: T1) => U1,
    func2: (value: T2) => U2,
    func3: (value: T3) => U3
): Variant<U1, U2, U3> {
    switch (variant.index) {
        case 0:
            return Variant.make1(func1(<T1>variant.value));
        case 1:
            return Variant.make2(func2(<T2>variant.value));
        case 2:
            return Variant.make3(func3(<T3>variant.value));
        default: throw new Error();
    }
}
```

3.4. Алгебраические типы данных

1. В — кортежи являются типами-произведениями.
2. Б — это тип-сумма языка TypeScript.
3. В — так как кортежи — это типы-произведения, количества возможных значений двух перечисляемых типов данных перемножаются (`2 x 3`).
4. Б — поскольку это тип-сумма, количества возможных значений двух перечисляемых типов данных складываются (`2 + 3`).

4 Типобезопасность

В этой главе

- Избегаем антипаттерна одержимости простыми типами данных.
- Обеспечиваем соблюдение ограничений при формировании экземпляров типов.
- Повышаем безопасность с помощью добавления информации о типе.
- Повышаем гибкость, скрывая и восстанавливая информацию о типе.

Вы уже знаете, как использовать основные типы данных, предоставляемые языком программирования, и как создавать новые типы путем их сочетания. Теперь посмотрим, как повысить безопасность наших программ с помощью типов данных. Под *безопасностью* я имею в виду уменьшение числа потенциальных ошибок.

Существует несколько способов добиться этой цели с помощью создания новых типов данных, кодирующих дополнительную информацию: смысловое содержание и гарантии. Кодирование смыслового содержания, о котором мы поговорим в первом разделе данной главы, не позволяет неправильно интерпретировать значение, например перепутать мили с километрами. А второе означает возможность кодирования в системе типов таких гарантий, как «экземпляр данного типа не может быть меньше `0`». Обе методики повышают безопасность кода, исключая из соответствующего типу множества возможных значений некорректные и позволяя избежать недоразумений как можно раньше, по возможности на этапе компиляции либо при создании объектов типов на этапе выполнения. При наличии экземпляра типа мы сразу знаем, что он собой представляет и что его значение — допустимое.

А раз уж мы говорим о типобезопасности, то обсудим также, как вручную добавлять и скрывать информацию от модуля проверки типов. И каким-то образом, обладая большей информацией, чем модуль проверки типов, мы можем попросить его довериться нам и передать ему эту информацию. С другой стороны, если модуль проверки типов знает слишком много и мешает нашей работе, то мы можем сделать так, что он «забудет» часть информации о типах, а это даст нам больше гибкости за счет безопасности. Некоторые из методик следует использовать с осторожностью, поскольку они делегируют обязанности должной проверки типов от модуля проверки типов нам как разработчикам. Но, как мы увидим далее, существуют вполне допустимые сценарии, при которых они необходимы.

4.1. Избегаем одержимости простыми типами данных, чтобы исключить неправильное толкование значений

В этом разделе я покажу, как использование для представления значений простых типов с неявными допущениями о том, что эти значения отражают, может вызывать проблемы. Например, когда эти допущения оказываются различными в двух отдельных (зачастую написанных разными разработчиками) частях кода (рис. 4.1).

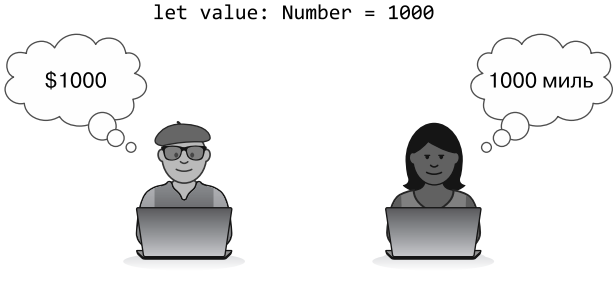


Рис. 4.1. Числовое значение 1000 может означать 1000 долларов или 1000 миль. Два разных разработчика могут интерпретировать его совершенно по-разному

Лучше положиться на систему типов и явно указать подобные допущения, описав соответствующие типы. В этом случае модуль проверки типов сможет выявить несоответствия и сообщить о них до того, как возникнет некая проблема.

Допустим, у нас есть функция `addToBill()`, принимающая в качестве аргумента `number`. Она должна прибавлять цену товара к счету. Поскольку тип аргумента — `number`, можно легко передать ей расстояние между городами в милях, также представленное в виде значения `number`. В итоге мы будем суммировать мили с долларами, а модуль проверки типов ничего даже не заподозрит!

С другой стороны, если наша функция `addToBill()` будет принимать аргумент типа `Currency`, а для расстояния между городами станет использоваться тип `Miles`, то код просто не скомпилируется (рис. 4.2).

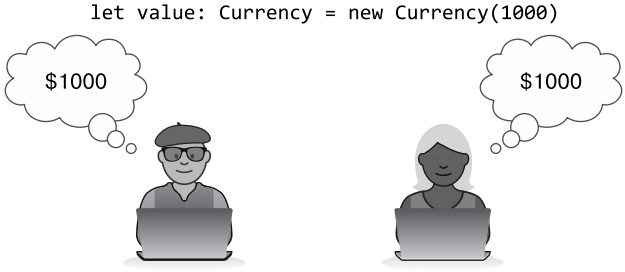


Рис. 4.2. Явное использование типа Currency четко указывает, что значение представляет собой не 1000 миль, а 1000 долларов

4.1.1. Аппарат Mars Climate Orbiter

Аппарат Mars Climate Orbiter развалился, поскольку разработанный компанией Lockheed компонент использовал не такую единицу измерения (фунт-силы на секунду, lbfs), которую ожидал другой компонент (ньютоны на секунду, Ns), разработанный НАСА. Попробуем представить, как мог выглядеть код этих двух компонентов. Функция `trajectoryCorrection()` ожидает измерение в ньютон-секундах (Н · с, метрическая единица измерения для импульса силы), а функция `provideMomentum()` выдает измерения в фунт-силах на секунду, как показано в листинге 4.1.

Листинг 4.1. Эскиз архитектуры с несовместимыми компонентами

```
function trajectoryCorrection(momentum: number) {
    if (momentum < 2 /* Н · с */)  {
        disintegrate();
    }

    /* ... */
}

function provideMomentum() {
    trajectoryCorrection(1.5 /* фунт-силы · с */);
}
```

Функция trajectoryCorrection() принимает значение импульса силы в виде аргумента типа number

Если импульс силы меньше 2 Н · с, то самоуничтожаемся

Функция provideMomentum передает измерение, равное 1,5 фунт-силы · с

В метрической системе единиц 1 фунт-сила · с равна 4,448222 Н · с. С точки зрения функции `provideMomentum()` передаваемое значение — допустимое, поскольку 1,5 фунт-силы · с равно более чем 6 Н · с, а это намного превышает нижний предел 2 Н · с. Что же не так? Основная проблема в данном случае такова: тип импульса силы в обоих компонентах — число с неявными допущениями относительно единицы измерения. Функция `trajectoryCorrection()` интерпретирует импульс силы как 1 Н · с, что меньше нижнего предела 2 Н · с, и ошибочно запускает самоуничтожение.

Посмотрим, как воспользоваться системой типов для предотвращения подобных катастрофических недоразумений. Сделаем единицы измерения явными, описав типы `Lbfs` и `Ns` в листинге 4.2. Оба типа служат обертками для числовых значений,

представляющих фактические значения величин. Мы укажем в каждом из типов уникальный символ, поскольку TypeScript считает типы одинаковой формы совместимыми, как мы увидим, когда будем говорить о создании подтипов. Благодаря трюку с уникальным символом неявная интерпретация одного типа как другого становится невозможной. Не во всех языках программирования необходим такой дополнительный уникальный символ. Подробнее мы поговорим об этом трюке в главе 7, а пока сосредоточим внимание на описании новых типов.

Листинг 4.2. Типы для единиц фунт-сила · с и Н · с

```
declare const NsType: unique symbol;

class Ns {
    readonly value: number;
    [NsType]: void;

    constructor(value: number) {
        this.value = value;
    }
}

declare const LbfsType: unique symbol;

class Lbfs {
    readonly value: number;
    [LbfsType]: void;

    constructor(value: number) {
        this.value = value;
    }
}
```

Таким специфическим для TypeScript образом гарантируется, что другие объекты аналогичной формы не будут интерпретироваться, как этот тип

Класс Ns по сути просто обертка для значения типа number

Аналогично тип Lbfs служит оберткой для значения типа number плюс уникальный символ

Теперь у нас есть два отдельных типа, и мы можем легко реализовать функции преобразования между ними, поскольку знаем, как они соотносятся друг с другом. Взглянем на листинг 4.3, в котором описана функция преобразования из фунт-сил · с в Н · с, необходимая для исправления кода нашей функции `trajectoryCorrection()`.

Листинг 4.3. Преобразование из фунт-сил · с в Н · с

```
function lbfsToNs(lbfs: Lbfs): Ns {
    return new Ns(lbfs.value * 4.448222);
}
```

Умножаем значение в фунт-силах · с на коэффициент преобразования и возвращаем значение в Н · с

Вернемся к аппарату Mars Climate Orbiter. Теперь мы можем переделать две функции, используя новые типы. Функция `trajectoryCorrection()` по-прежнему ожидает в качестве аргумента импульс силы в Н · с (и произведет самоуничтожение аппарата, если значение окажется меньше 2 Н·с), а функция `provideMomentum()` все еще выдает измерения в фунт-силах · с. Но теперь уже нельзя просто передать функции `trajectoryCorrection()` значение, возвращенное функцией `provideMomentum()`, поскольку типы возвращаемого ей значения и аргумента функции различны.

Придется явно провести преобразование из одного типа в другой с помощью нашей функции `lbfsToNs()`, как показано в листинге 4.4.

Листинг 4.4. Модифицированные компоненты

```
function trajectoryCorrection(momentum: Ns) {
    if (momentum.value < new Ns(2).value)  {
        disintegrate();
    }

    /* ... */
}

function provideMomentum() {
    trajectoryCorrection(lbfsToNs(new Lbfs(1.5)));
}
```

Функция trajectoryCorrection() теперь получает аргумент типа Ns и сравнивает его со значением 2 Н · с

Функция provideMomentum выдает значение 1,5 фунт-силы · с и должна преобразовать его в Н · с

Если опустить преобразование `lbfsToNs()`, то код просто не скомпилируется и мы получим следующую ошибку: `Argument of type 'lbfs' is not assignable to parameter of type 'Ns'. Property '[NsType]' is missing in type 'lbfs'` (Невозможно присвоить аргумент типа `'lbfs'` параметру типа `Ns`. В типе `'lbfs'` отсутствует свойство `'[NsType]'`).

Разберемся, что произошло. Сначала у нас было два компонента, работавших со значениями импульса силы, и, хоть они взаимодействовали с разными единицами измерения, оба представляли значения просто в виде `number`. Во избежание неправильной интерпретации мы создали несколько новых типов, по одному для каждой единицы измерения, благодаря чему исключили всякую возможность неправильной интерпретации. Если компонент явным образом работает с типом `Ns`, то он не может случайно воспользоваться значением типа `lbfs`.

Отмечу также, что описанные в виде комментариев в первом нашем примере допущения (`1.5 /* lbfs */`) в итоговой реализации превратились в код (`new Lbfs(1.5)`).

4.1.2. Антипаттерн одержимости простыми типами данных

Паттерны проектирования отражают весьма надежные и эффективные архитектурные элементы программного обеспечения, допускающие повторное использование. Аналогично этому антипаттерны представляют собой часто встречающиеся архитектурные элементы, которые являются неэффективными и зачастую приводят к обратному результату и для которых существует лучшая альтернатива. Вышеупомянутый пример — образец известного антипаттерна под названием *«одержимость простыми типами данных»* (primitive obsession). Такая одержимость проявляется, когда базовые типы данных используются для всего, что только можно: тип `number` для почтового индекса, `string` — для телефонного номера и т. д.

Попадание в эту ловушку может вызвать множество различных ошибок, таких как представленная в предыдущем подразделе. Причина в том, что смысл значений не отражается явным образом в системе типов. Если разработчик получает значение импульса силы в виде `number`, то может неявно предполагать, что оно представляет

собой значение в Н · с. У модуля проверки типа недостаточно информации, чтобы обнаружить несовместимые допущения двух разных разработчиков. Если же подобное допущение явным образом описывается в виде объявления типа и разработчик получает значение импульса силы в виде экземпляра `Ns`, то модуль проверки типов может проверить, не пытается ли другой разработчик передать вместо него экземпляр `Lbfs` и оборвать компиляцию такого кода.

И хотя почтовый индекс представляет собой число, хранить его в виде значения типа `number` не стоит. Импульс силы никогда не должен интерпретироваться как почтовый индекс.

Что касается представления простых сущностей, например результатов измерения физических величин и почтовых индексов, то их можно описать в виде новых типов данных, даже если эти типы окажутся простыми адаптерами для чисел или строк. Благодаря этому система типов получит больше информации для анализа нашего кода и мы исключим целый класс ошибок, вызванных несовместимыми допущениями, не говоря уже о повышении читабельности кода. Сравните, например, первое определение функции `trajectoryCorrection()`, а именно `trajectoryCorrection(momentum: number)`, со вторым, `trajectoryCorrection(momentum: Ns)`. Из второго разработчик, читающий код, может получить больше информации о принятых допущениях (о том, что ожидается импульс силы в Н · с).

До сих пор мы обертывали простые типы данных в другие типы, чтобы закодировать больше информации. Теперь перейдем к повышению безопасности с помощью ограничения диапазона допустимых значений заданного типа.

4.1.3. Упражнение

Каков наиболее безопасный способ выразить измеренный вес?

А. В виде `number`.

Б. В виде `string`.

В. В виде пользовательского типа `Kilograms`.

Г. В виде пользовательского типа `Weight`.

4.2. Обеспечиваем соблюдение ограничений

В главе 3 мы говорили о сочетании типов и объединении основных типов данных для представления более сложных понятий, например представления точки на двумерной плоскости в виде пары числовых значений по одному для каждой из координат x и y. Теперь посмотрим, что делать, если диапазон значений готового типа данных больше, чем требуется.

Возьмем, например, результат измерения температуры. Мы хотели бы избежать одержимости простыми типами данных, поэтому объявим тип `Celsius`, чтобы четко указать, в каких именно единицах должна измеряться температура. Этот тип также будет просто оберткой для числа.

Впрочем, существует дополнительное ограничение: температура не должна опускаться ниже абсолютного нуля, равного –273,15 градуса Цельсия. Одна из возмож-

ностей — проверять допустимость значения при каждом использовании экземпляра данного типа. Впрочем, в процессе могут возникать ошибки: мы всегда добавляем проверку, а пришедший в команду новый разработчик не знает принятого паттерна и этого не сделает. Не лучше ли устроить так, чтобы получить некорректное значение было просто нельзя?

Сделать это можно двумя способами: с помощью конструктора или фабрики.

4.2.1. Обеспечиваем соблюдение ограничений с помощью конструктора

Можно реализовать ограничение в конструкторе и поступить со слишком маленьким значением одним из двух способов, которые мы использовали при переполнении целых чисел. Первый способ — сгенерировать в случае некорректного значения исключение и запретить создание объекта (листинг 4.5).

Листинг 4.5. Генерация конструктором исключения в случае некорректного значения

```
declare const celsiusType: unique symbol;

class Celsius {
    readonly value: number;
    [celsiusType]: void;

    constructor(value: number) {
        if (value < -273.15) throw new Error();

        this.value = value;
    }
}
```

Значение неизменяемое, поэтому после начальной инициализации поменять его нельзя

Конструктор генерирует исключение, если попытаться создать объект с некорректным значением температуры

Сделав значение `readonly`, мы гарантируем, что оно останется корректным после формирования. Можно также сделать его приватным и обращаться к нему с помощью функции-геттера (чтобы его можно было получить, но не задать).

Можно также реализовать конструктор так, чтобы он делал значение допустимым: превращал любое значение меньше `-273.15` в `-273.15` (листинг 4.6).

Листинг 4.6. Конструктор, исправляющий некорректное значение

```
declare const celsiusType: unique symbol;

class Celsius {
    readonly value: number;
    [celsiusType]: void;

    constructor(value: number) {
        if (value < -273.15) value = -273.15;

        this.value = value;
    }
}
```

Вместо генерации исключения мы «исправляем» значение

Оба эти подхода допустимы в зависимости от сценария. Можно также воспользоваться функцией-фабрикой. *Фабрика* (factory) — это класс (функция), основная задача которого состоит в создании другого объекта.

4.2.2. Обеспечиваем соблюдение ограничений с помощью фабрики

Фабрика удобна в случае, если желательно не генерировать исключение, а вернуть `undefined` либо какое-то другое значение (не температуру), которое бы указывало на невозможность создания корректного экземпляра. Конструктор не может сделать это, поскольку не возвращает значений: он либо производит инициализацию экземпляра, либо генерирует исключение. Еще одна причина использовать фабрику — сложная логика формирования и проверки объекта, из-за чего имеет смысл реализовать ее вне конструктора. Примите как эмпирическое правило, что конструкторы не должны производить сложных вычислений, а только задавать начальные значения членов объекта.

Рассмотрим реализацию фабрики в листинге 4.7. Мы сделаем конструктор приватным, чтобы его мог вызывать только фабричный метод. Фабрика будет статическим методом класса и станет возвращать экземпляр класса `Celsius` или `undefined`.

Листинг 4.7. Фабрика, возвращающая undefined в случае некорректного значения

```
declare const celsiusType: unique symbol;

class Celsius {
    readonly value: number;
    [celsiusType]: void;

    private constructor(value: number) {
        this.value = value;
    }

    static makeCelsius(value: number): Celsius | undefined {
        if (value < -273.15) return undefined;

        return new Celsius(value);
    }
}
```

Конструктор теперь становится приватным, поскольку не производит сам никаких проверок

Фабрика возвращает экземпляр класса Celsius или undefined

Фабрика — единственный способ создания экземпляров класса Celsius — обеспечивает соблюдение ограничения

Во всех этих случаях мы получаем дополнительную гарантию, что значение экземпляра класса `Celsius` никогда не окажется меньше `-273.15`. Преимущество проверки при создании экземпляра типа и невозможности создания экземпляра другим способом состоит в полной гарантии допустимого значения любого передаваемого экземпляра данного типа.

Вместо проверки допустимости экземпляра при его использовании, что обычно означает проверку во многих местах кода, мы производим эту проверку только

в одном месте с гарантией того, что некорректный объект типа просто не может существовать.

Конечно, эта методика выходит далеко за рамки создания простых оберток для значений, таких как класс `Celsius`. Можно обеспечить, например, корректность объекта `Date`, созданного на основе значений для года, месяца и дня, и запретить значения наподобие `31 июня`. Существует множество случаев, когда имеющиеся в нашем распоряжении базовые типы данных не позволяют напрямую наложить нужные нам ограничения, и мы можем создать типы, которые инкапсулируют дополнительные ограничения и гарантируют невозможность существования их экземпляров с некорректными значениями.

Далее мы поговорим о добавлении и сокрытии информации о типах в нашем коде и сценариях, при которых это может принести пользу.

4.2.3. Упражнение

Реализуйте тип `Percentage`, отражающий значение от `0` до `100`. Значения меньше `0` должны преобразовываться в `0`, а значения больше `100` — в `100`.

4.3. Добавляем информацию о типе

Несмотря на серьезный теоретический фундамент проверки типов, во всех языках программирования существуют способы обхода проверок типов, позволяющие дать команду компилятору рассматривать значение как относящееся к определенному типу. Фактически мы говорим компилятору: «Доверься нам; мы знаем, что это за тип, лучше, чем ты». Это называется *приведением типов* (type cast) — термин, который вы наверняка уже слышали.

ПРИВЕДЕНИЕ ТИПОВ

Приведение типов означает преобразование типа выражения в другой тип. У всех языков программирования свои правила относительно того, какие преобразования допустимы, а какие — нет, какие компилятор может произвести автоматически, а для каких необходимо писать дополнительный код (рис. 4.3).

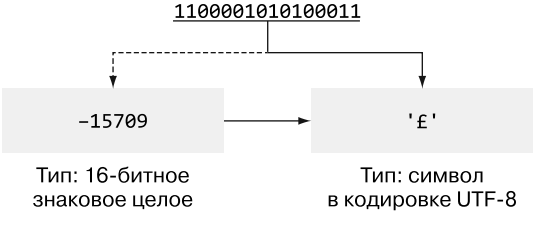


Рис. 4.3. С помощью приведения типов можно превратить 16-битное знаковое целое в символ в кодировке UTF-8

4.3.1. Приведение типов

Явным (explicit type cast) называется такое приведение типов, при котором разработчик явно указывает компилятору рассматривать значение как относящееся к определенному типу. В TypeScript приведение к типу `NewType` производится с помощью добавления `<NewType>` перед значением или `as NewType` после него.

Неправильное применение этой методики может быть опасным: при обходе модуля проверки типа можно получить ошибку во время выполнения, если попытаться использовать значение в качестве чего-то, чем оно не является. Например, я могу привести мой `Bike` (велосипед) с функцией `ride()` к типу `SportsCar`, но у него все равно не появится функция `drive()`[1], как видно из листинга 4.8.

Листинг 4.8. Ошибка во время выполнения в результате приведения типов

```
class Bike {
    ride(): void {/* ... */ }
}

class SportsCar {
    drive(): void {/* ... */ }
}

let myBike: Bike = new Bike();

myBike.ride();

let myPretendSportsCar: SportsCar = <SportsCar><unknown>myBike;

myPretendSportsCar.drive();
```

Объект myBike создан как объект типа Bike, поэтому у него есть функция ride()

Говорим компилятору, чтобы считал myBike объектом типа SportsCar, и присваиваем его переменной myPretendSportsCar

Попытка вызвать функцию drive() объекта myPretendSportsCar приводит к ошибке во время выполнения

Мы можем потребовать от модуля проверки типа притвориться, будто наш объект — `SportsCar`, но это не значит, что он действительно представляет собой спортивный автомобиль. Вызов функции `drive()` приводит к генерации следующего исключения: `TypeError: myPretendSportsCar.drive is not a function` (Ошибка типа: `myPretendSportsCar.drive` не является функцией).

Нам пришлось сначала привести `myBike` к типу `unknown` и лишь затем к типу `SportsCar`, поскольку компилятор понимает, что типы `Bike` и `SportsCar` не перекрываются (допустимое значение одного из них не может быть допустимым значением другого). Как следствие, простой вызов `<SportsCar>myBike` приводит к ошибке. Вместо этого нам приходится сначала произвести приведение `<unknown>Bike`, чтобы компилятор «забыл» тип переменной `myBike`. И только потом мы можем сказать ему: «Доверься нам, это `SportsCar`». Но, как мы видим, в результате все равно происходит

[1] Автор обыгрывает тот факт, что в английском языке для описания езды на велосипеде используется глагол to ride, а для езды на автомобиле — to drive. — *Примеч. пер.*

ошибка во время выполнения. В других языках программирования это привело бы к фатальному сбою программы. Как правило, подобная ситуация недопустима. Так когда же может пригодиться приведение типов?

4.3.2. Отслеживание типов вне системы типов

Иногда мы знаем о типе больше, чем модуль проверки типов. Вновь обратимся к реализации типа `Either` из главы 3. В нем содержится значение типа `TLeft` или `TRight`, а также флаг типа `boolean`, отслеживающий, относится ли хранимое значение к типу `TLeft`, как показано в листинге 4.9.

Листинг 4.9. Возвращаемся к реализации типа Either

```
class Either<TLeft, TRight> {
    private readonly value: TLeft | TRight;
    private readonly left: boolean;

    private constructor(value: TLeft | TRight, left: boolean) {
        this.value = value;
        this.left = left;
    }

    isLeft(): boolean {
        return this.left;
    }

    getLeft(): TLeft {
        if (!this.isLeft()) throw new Error();

        return <TLeft>this.value;
    }

    isRight(): boolean {
        return !this.left;
    }

    getRight(): TRight {
        if (!this.isRight()) throw new Error();

        return <TRight>this.value;
    }

    static makeLeft<TLeft, TRight>(value: TLeft) {
        return new Either<TLeft, TRight>(value, true);
    }

    static makeRight<TLeft, TRight>(value: TRight) {
        return new Either<TLeft, TRight>(value, false);
    }
}
```

Храним значение типа TLeft или Tright

Отслеживаем, относится ли хранимое значение к типу TLeft, с помощью свойства left

Если нужно получить TLeft, то проверяем, хранится ли в объекте нужный тип данных, а затем приводим значение к типу Tleft

Фабрика makeLeft задает начальное значение свойства left, равное true; makeRight — false

Таким образом, мы объединяем два типа в тип-сумму, с помощью которого можно представить значение любого из них. Впрочем, если посмотреть внимательнее, то окажется, что тип хранимого значения — `TLeft | TRight`. После присваивания модуль проверки типов больше не знает, относилось ли фактически хранимое значение к типу `TLeft` или `TRight`. Начиная с этого момента он будет считать, что `value` может относиться к любому из них. Именно это нам и требуется. Однако в какой-то момент мы захотим воспользоваться данным значением.

Компилятор не разрешит нам передать значение типа `TLeft | TRight` функции, ожидающей значение типа `TLeft`: если наше значение на самом деле окажется `TRight`, то это приведет к проблемам. Треугольник *или* квадрат не всегда можно протащить через треугольное отверстие. Пройти через него сможет только треугольник. Но что, если фактическим значением окажется квадрат (рис. 4.4)?

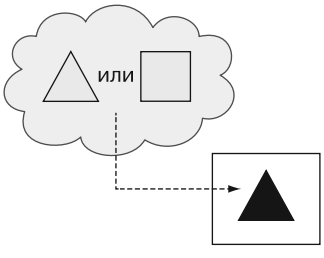


Рис. 4.4. Если наш объект представляет собой треугольник или квадрат, то мы не можем сказать наверняка, пройдет ли фактическое значение через треугольное отверстие. Объект-треугольник пройдет, объект-квадрат — нет

Подобная попытка приведет к ошибке компиляции, что хорошо. Однако нам известно то, чего не знает модуль проверки типов: с момента установки значения мы знаем, к какому типу оно относится: `TLeft` или `TRight`. Если мы создали объект с помощью метода `makeLeft()`, то свойство `left` равно `true`. Если же мы создали объект с помощью `makeRight()`, то это свойство равно `false`, как показано в листинге 4.10. Мы отслеживаем это, даже если модуль проверки типов забывает.

Листинг 4.10. Методы makeLeft() и makeRight()

```
class Either<TLeft, TRight> {
    private readonly value: TLeft | TRight;
    private readonly left: boolean;

    private constructor(value: TLeft | TRight, left: boolean) {
        this.value = value;
        this.left = left;
    }

    /* ... */

    static makeLeft<TLeft, TRight>(value: TLeft) {
        return new Either<TLeft, TRight>(value, true);
    }

    static makeRight<TLeft, TRight>(value: TRight) {
        return new Either<TLeft, TRight>(value, false);
    }
}
```

Свойство left указывает, хранится ли в объекте Tleft

Значение left задается в приватном конструкторе, который могут вызывать только методы makeLeft() и makeRight()

Методы makeLeft() и makeRight() устанавливают конкретное значение свойства left

Извлекая значение, вызывающая сторона должна сначала проверить, к какому из двух типов оно фактически относится. Если мы работаем с типом `Either<Triangle, Square>` и хотим получить `Triangle`, то сначала должны вызвать `isLeft()`. В случае

возврата `true` мы можем вызвать `getLeft()` и получить объект `Triangle`, как показано в листинге 4.11.

Листинг 4.11. Triangle или Square

```
declare const triangleType: unique symbol;
class Triangle {
    [triangleType]: void;
    /* ... */
}

declare const squareType: unique symbol;
class Square {
    [squareType]: void;
    /* ... */
}

function slot(triangle: Triangle) {
    /* ... */
}

let myTriangle: Either<Triangle,Square>
    = Either.makeLeft(new Triangle());

if (myTriangle.isLeft())
    slot(myTriangle.getLeft());
```

Типы Triangle и Square

С этого момента тип myTriangle.value — Triangle | Square. Компилятор больше не знает, что мы поместили туда Triangle

Вызов метода getLeft() приводит значение обратно к типу Triangle

Внутри нашей реализации `getLeft()` производятся все необходимые проверки (в данном случае проверяется, что `this.isLeft()` равно `true`) и обработка недопустимых вызовов (в этом случае генерируется `Error`). После всего этого значение приводится к нужному типу. Модуль проверки типов уже «забыл», какой тип был у значения при присваивании, поэтому мы ему напоминаем, как показано в листинге.4.12, ведь мы отслеживали тип с помощью свойства `left`.

Листинг 4.12. isLeft() и getLeft()

```
class Either<TLeft, TRight> {
    private readonly value: TLeft | TRight;
    private readonly left: boolean;

    /* ... */

    isLeft(): boolean {
        return this.left;
    }

    getLeft(): TLeft {
        if (!this.isLeft()) throw new Error();

        return <TLeft>this.value;
    }

    /* ... */
}
```

Клиенты могут проверять, относится ли значение к типу TLeft, с помощью вызова метода isLeft()

Если значение не того типа, то обрабатываем ошибку. В данном случае мы генерируем Error. Можно было бы также вернуть undefined

Значение приводится к типу TIeft

В данном случае операция приведения к типу `<unknown>` нам не нужна: значение типа `TLeft | TRight` вполне может быть допустимым значением типа `TLeft`, так что компилятор не станет жаловаться и доверится указанному нами приведению типа.

Приведение типов, использованное должным образом, дает большие возможности, позволяя уточнять тип значения. При наличии объекта типа `Triangle | Square`, о котором известно, что на самом деле он представляет собой `Triangle`, можно привести его к типу `Triangle`, который компилятор позволит протащить через треугольное отверстие.

На самом деле большинство модулей проверки типов производят подобные приведения типов автоматически и никакого кода для этого писать не требуется.

НЕЯВНОЕ И ЯВНОЕ ПРИВЕДЕНИЕ ТИПОВ

Неявное приведение типов (implicit type cast, coercion) производится компилятором автоматически. Оно не требует написания никакого кода. Подобное приведение типов обычно безопасно. Явное приведение типов, напротив, требует написания кода. Оно фактически обходит правила системы типов, и использовать его следует с осторожностью.

4.3.3. Распространенные разновидности приведения типов

Рассмотрим несколько распространенных видов приведения типов, как явных, так и неявных, и выясним, в каких случаях они могут пригодиться.

Понижающее и повышающее приведение типов

Один из часто встречающихся примеров приведения типов — интерпретация объекта типа, унаследованного от другого, как объекта родительского типа. Если класс `Triangle` унаследован от базового класса `Shape`, то объект `Triangle` можно использовать везде, где требуется `Shape`, как показано в листинге 4.13.

Внутри тела метода `useShape()` компилятор рассматривает его аргумент как `Shape`, даже если мы передали `Triangle`. Интерпретация унаследованного класса (`Triangle`) как базового (`Shape`) называется *повышающим приведением типов* (upcast). Если мы уверены, что наша фигура фактически является треугольником, то можем привести ее обратно к типу `Triangle`, но такое приведение типа должно быть описано явно. Приведение от родительского класса к унаследованному, показанное в листинге 4.14, называется *понижающим приведением типа* (downcast), и большинство строго типизированных языков программирования автоматически его не производят.

В отличие от повышающего приведения типов, понижающее небезопасно. Хотя по унаследованному классу сразу понятно, каков его родительский класс, компилятор не может автоматически определить по родительскому, к какому из возможных унаследованных классов относится значение.

Листинг 4.13. Понижающее приведение типа

```
class Shape {
    /* ... */
}

declare const triangleType: unique symbol;

class Triangle extends Shape {
    [triangleType]: void;
    /* ... */
}

function useShape(shape: Shape) {
    /* ... */
}

let myTriangle: Triangle = new Triangle();

useShape(myTriangle);
```

Тип Triangle расширяет тип Shape

Метод useShape() ожидает аргумент типа Shape

Мы можем передать ему объект типа Triangle, и он будет автоматически приведен к типу Shape

Некоторые языки программирования хранят дополнительную информацию о типе на этапе выполнения и содержат оператор `is`, позволяющий выяснить тип объекта. При создании нового объекта вместе с ним хранится его тип, так что, даже если с помощью понижающего приведения скрыть от компилятора часть информации о типе, на этапе выполнения можно будет проверить, является ли наш объект экземпляром конкретного типа, задействовав `if (shape is Triangle)...`.

Листинг 4.14. Понижающее приведение типов

```
class Shape {
    /* ... */
}

declare const triangleType: unique symbol;

class Triangle extends Shape {
    [triangleType]: void;
    /* ... */
}

function useShape(shape: Shape, isTriangle: boolean) {
    if (isTriangle) {   #B
        let triangle: Triangle = <Triangle>shape;
        /* ... */
    }
    /* ... */
}

let myTriangle: Triangle = new Triangle();

useShape(myTriangle, true);
```

У этой версии функции есть дополнительный аргумент для отслеживания того, был ли передан треугольник

Если аргумент фактически представляет собой треугольник, то мы можем выполнить обратное приведение типа

Вызывающая сторона должна правильно задать значение этого флага; в противном случае произойдет ошибка во время выполнения

Языки и среды выполнения, в которых реализована подобная информация о типе времени выполнения, обеспечивают безопасный способ хранения и запроса информации о типах, так что риска рассогласования данной информации с объектами нет. Конечно, это означает определенные затраты на хранение в памяти дополнительной информации о каждом экземпляре класса.

В главе 7, когда мы будем обсуждать создание подтипов, рассмотрим более сложные случаи повышающего приведения типов и поговорим о вариантности. А пока перейдем к обсуждению расширяющего и сужающего приведения типов.

Расширяющее и сужающее приведение типов

Еще один распространенный вид неявного приведения типов — из целочисленного типа данных с фиксированным числом битов, скажем, восьмибитного беззнакового целого — в другой целочисленный тип данных, представляющий значения с помощью большего числа битов, — например, 16-битное беззнаковое целое. Такое приведение типов называется *расширяющим* (widening cast).

С другой стороны, приводить знаковое целое к беззнаковому опасно, поскольку беззнаковое не позволяет отражать отрицательные числа. Аналогично приведение целочисленного типа данных с большим числом битов к типу с меньшим, например 16-битного знакового целого к восьмибитному знаковому целому, подходит только для чисел, которые можно представить с помощью этого меньшего типа.

Подобное приведение типов называется *сужающим* (narrowing cast). Некоторые компиляторы требуют описания такого приведения явным образом в силу его небезопасности. Явное приведение полезно, поскольку ясно демонстрирует, что разработчик не сделал этого ненамеренно. Некоторые другие компиляторы разрешают сужающее приведение типов, но выдают предупреждение. Поведение на этапе выполнения, если значение не помещается в новом типе данных, аналогично целочисленному переполнению, которое мы обсуждали в главе 2: в зависимости от языка программирования возвращается ошибка или значение усекается так, чтобы поместиться в новый тип данных (рис. 4.5).

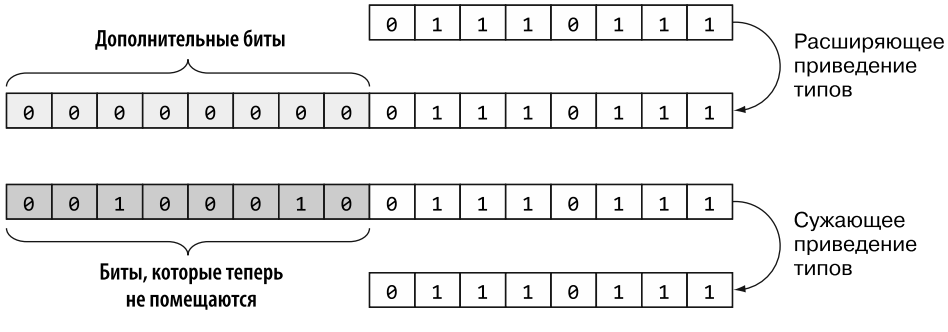


Рис. 4.5. Пример расширяющего и сужающего приведения типов. Расширяющее безопасно: серые прямоугольники отражают полученные дополнительные биты, так что информация не потеряется. И наоборот, сужающее небезопасно: черные прямоугольники соответствуют битам, не помещающимся в новый тип данных

Приведение типов следует использовать осмотрительно, поскольку оно обходит модуль проверки типов, фактически сводя на нет все преимущества проверки типов. Впрочем, это удобные инструменты, особенно если у нас больше информации, чем у компилятора, и мы хотели бы сообщить ее ему. Получив ее от нас, компилятор может использовать эту информацию при дальнейшем анализе. Возвращаясь к примеру с `Triangle | Square`: если компилятору указано, что значение представляет собой `Triangle`, оно уже нигде не будет фигурировать как `Square`. Эта методика аналогична описанной в разделе 4.2, в котором мы обсуждали обеспечение соблюдения ограничений, но в данном случае вместо проверки на этапе выполнения мы просто предлагаем компилятору довериться нам.

В следующем разделе мы рассмотрим несколько других ситуаций, в которых удобно будет заставить компилятор «забыть» информацию о типе.

4.3.4. Упражнения

1. Какой из следующих видов приведения типов считается безопасным?
   А. Повышающее приведение типов.
   Б. Понижающее приведение типов.
   В. Повышающее и понижающее приведение типов.
   Г. Ни то ни другое.
2. Какой из следующих видов приведения типов считается небезопасным?
   А. Повышающее приведение типов.
   Б. Понижающее приведение типов.
   В. Повышающее и понижающее приведение типов.
   Г. Ни то ни другое.

4.4. Скрываем и восстанавливаем информацию о типе

Один из примеров того, когда может понадобиться скрывать информацию о типе: необходимость создания коллекции, содержащей набор значений различных типов. Если коллекция содержит значения лишь одного типа, как мешок с кошками, то все просто, поскольку нам известно: что бы мы ни вытащили из мешка — это будет кошка. Если же положить в мешок еще и некие продукты питания, то мы можем вытащить либо кошку, либо какой-то продукт (рис. 4.6).

Коллекция, содержащая элементы одного типа, подобно нашему мешку с кошками, называется *однородной* (homogenous collection). Скрывать информацию о типе элементов нет смысла, поскольку тип у них одинаков. Коллекция элементов различных типов называется *неоднородной* (heterogeneous collection). Для объявления подобной коллекции необходимо скрыть часть информации о типе.

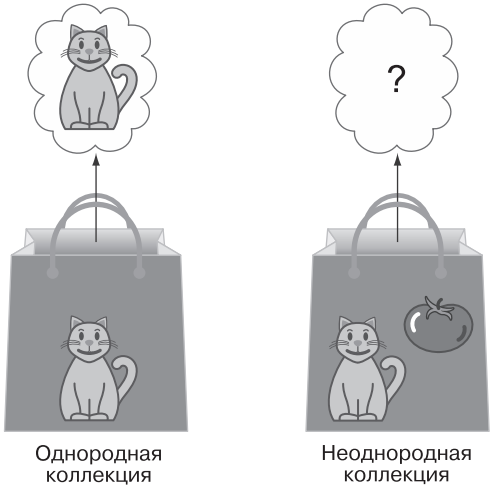


Рис. 4.6. Если в мешке содержатся только кошки, то можно биться о заклад: что бы мы ни вытащили из мешка — это будет кошка. Если же в мешке лежат еще и продукты питания, то мы уже не можем знать, что именно вытащим

4.4.1. Неоднородные коллекции

Документ может содержать текст, изображения и таблицы. При работе с документом желательно хранить все составляющие его элементы вместе, так что мы будем хранить их в какой-нибудь коллекции. Но какого типа должны быть ее элементы? Существует несколько способов реализовать это, и все они требуют сокрытия какой-либо информации о типе.

Базовый класс (интерфейс)

Можно создать иерархию классов и считать, что все элементы документа обязаны быть частью какой-либо иерархии. Если они все относятся к типу `DocumentItem`, то мы можем хранить коллекцию значений `DocumentItem`, даже если добавляем в коллекцию элементы таких типов, как `Paragraph`, `Picture` и `Table`. Аналогично можно объявить интерфейс `IDocumentItem`, и массив будет содержать только элементы типов, реализующих этот интерфейс, как показано в листинге 4.15.

Мы скрыли часть информации о типах, поэтому больше не знаем, относится ли конкретный элемент коллекции к типу `Paragraph`, `Picture` или `Table`, но знаем, что он реализует контракт `DocumentItem` или `IDocumentItem`. Если нам требуется только заданное данным контрактом поведение, то можно работать с элементами коллекции как с элементами этого общего типа. Если же нам нужен конкретный тип, например при необходимости передать изображение в плагин редактирования изображений, то придется произвести понижающее приведение `DocumentItem` или `IDocumentItem` к типу `Picture`.

Листинг 4.15. Коллекция типов, реализующих интерфейс IDocumentItem

```
interface IDocumentItem {
    /* ... */
}

class Paragraph implements IDocumentItem {
    /* ... */
}

class Picture implements IDocumentItem {
    /* ... */
}

class Table implements IDocumentItem {
    /* ... */
}

class MyDocument {
    items: IDocumentItem[];

    /* ... */
}
```

IDocumentItem — интерфейс, общий для всех элементов документа

Каждый из классов Paragraph, Picture и Table реализует интерфейс IDocumentItem

Мы храним элементы документа как массив объектов типа IDocumentItem

Тип-сумма или вариантный тип данных

Если заранее знать все типы, с которыми придется иметь дело, то можно воспользоваться типом-суммой, как показано в листинге 4.16. Можно описать наш документ как массив `Paragraph | Picture | Table` (в этом случае придется отслеживать тип каждого элемента коллекции с помощью каких-либо дополнительных средств) или как вариантный тип данных `Variant<Paragraph, Picture, Table>` (имеющий внутренний механизм отслеживания хранимых типов).

Листинг 4.16. Коллекция типов как тип-сумма

```
class Paragraph {
    /* ... */
}

class Picture {
    /* ... */
}

class Table {
    /* ... */
}

class MyDocument {
    items: (Paragraph | Picture | Table)[];

    /* ... */
}
```

Классы Paragraph, Picture и Table больше не реализуют интерфейс

Коллекция элементов документа теперь представляет собой массив объектов, которые могут быть любым из этих типов

Оба способа: и `Paragraph | Picture | Table`, и `Variant<Paragraph, Picture, Table>` — позволяют хранить набор элементов, которые могут не иметь ничего общего (никакого общего базового типа или реализуемого интерфейса). Преимущество таких подходов — отсутствие требований к типам в коллекции. Недостаток — с элементами списка мало что можно сделать без приведения их к их фактическому типу или, в случае `Variant`, без вызова метода `visit()`, требующего написания соответствующей функции для каждого из возможных типов коллекции.

Напомню: поскольку тип наподобие `Variant` содержит информацию о том, какие типы фактически в нем хранятся, он знает, какую функцию выбрать из набора передаваемых в метод `visit()` функций.

Тип unknown

В самом крайнем случае коллекция может содержать что угодно. Как показано в листинге 4.17, в языке TypeScript есть специальный тип `unknown`, служащий для представления подобных коллекций. В большинстве объектно-ориентированных языков программирования существует общий базовый тип, родительский для всех остальных типов, обычно называемый `Object`. Мы рассмотрим этот вопрос подробнее в главе 7, когда будем обсуждать создание подтипов.

Листинг 4.17. Коллекция элементов типа unknown

```
class MyDocument {
    items: unknown[];
    /* ... */
}
```

Массив может содержать элементы любого типа

Благодаря данной методике наш документ может содержать что угодно. У типов не обязательно должен быть общий контракт, нам даже не требуется знать заранее, чем являются эти типы. С другой стороны, с элементами такой коллекции мало что можно сделать. Практически всегда придется приводить их к другим типам, вследствие чего необходимо отдельно отслеживать их исходные типы.

В табл. 4.1 приведена краткая сводка различных подходов, а также их достоинств и недостатков.

Таблица 4.1. За и против реализаций неоднородных списков

Подходы	Достоинства	Недостатки
Иерархия	Возможность использования любых свойств и методов базового типа без приведения типов	Содержащиеся в коллекции типы должны быть связаны через базовый тип или реализуемый интерфейс
Тип-сумма	Типы могут быть никак не связаны	Для использования элементов необходимо привести их обратно к фактическому типу, если метода visit() для вариантного типа данных нет
Тип unknown	Возможность хранить что угодно	Для использования элементов необходимо отслеживать их фактические типы и приводить обратно к этим типам

У всех этих примеров есть достоинства и недостатки. Все зависит от того, насколько гибкой должна быть наша коллекция в смысле хранимых элементов и насколько часто нам потребуется восстанавливать исходные типы элементов. Тем не менее все указанные примеры включают сокрытие какой-либо части информации о типе при помещении элементов в коллекцию. Еще один пример сокрытия и восстановления информации о типе — сериализация.

4.4.2. Сериализация

При записи информации в файл для последующей выгрузки ее обратно и использования в программе или при подключении к интернет-сервису с отправкой/получением каких-либо данных данные перемещаются в виде последовательностей битов. Сериализация (serialization) — это процесс кодирования значения определенного типа в виде последовательности битов. Обратная операция, *десериализация* (deserialization), представляет собой декодирование последовательности битов в структуру данных, с которой можно работать (рис. 4.7).

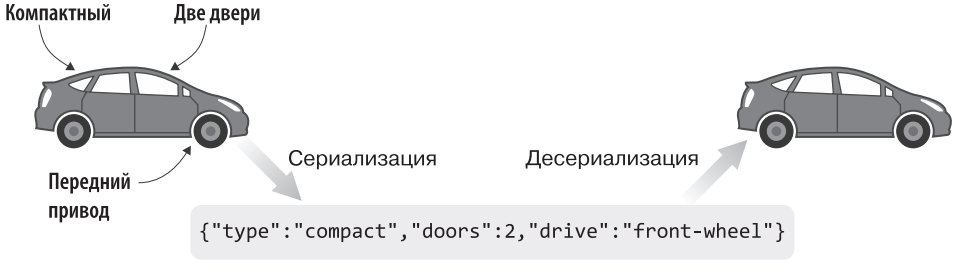


Рис. 4.7. Компактный автомобиль с двумя дверями и передним приводом, сериализованный в JSON, а затем десериализованный обратно в автомобиль

Конкретное кодирование зависит от используемого протокола, которым может быть JSON, XML или любой другой из множества доступных протоколов. С точки зрения типов важно то, что после сериализации мы получаем значение, эквивалентное исходному типизированному, однако никакая информация о системе типов теперь не доступна. По сути, мы получаем строку или массив байтов. Метод `JSON.stringify()` принимает в качестве аргумента объект и возвращает JSON-представление этого объекта в виде строки. Если преобразовать в строку объект `Cat` (Кошка), как показывает листинг 4.18, то можно записать результат на диск, отправить по сети или даже вывести на экран, но вызвать для него метод `meow()` (от англ. meow — «мяукать») не получится.

Мы по-прежнему знаем, что представляет собой значение, а модуль проверки типов — нет. Обратная операция означает преобразование сериализованного объекта обратно в типизированное значение. В данном случае мы можем воспользоваться методом `JSON.parse()`, который принимает на входе строку и возвращает объект JavaScript. А поскольку этот способ работает для произвольной строки, результат вызова имеет тип `any`.

Листинг 4.18. Сериализация кошки[1]

```
class Cat {
    meow() {
        /* ... */
    }
}

let serializedCat: string = JSON.stringify(new Cat());

// serializeCat.meow();
```

В типе Cat есть метод meow()

Мы сериализуем объект типа Cat в строку JSON с помощью метода JSON.stringify()

Понятно, что мы не можем использовать метод meow(), поскольку serializeCat представляет собой строку

ТИП ANY

TypeScript предоставляет разработчикам тип any, используемый для взаимодействия с JavaScript в случае недоступности информации о типе. Этот тип небезопасен, поскольку компилятор не производит проверки типа для его экземпляров, которые могут свободно преобразовываться в любой тип и из любого типа. Защита от неправильной интерпретации в этом случае ложится на плечи разработчика.

Если нам известно, что имеющийся у нас объект представляет собой сериализованный объект `Cat`, то мы можем присвоить его новому объекту `Cat` с помощью метода `Object.assign()`, как показано в листинге 4.19, приведя его затем обратно к исходному типу, поскольку `Object.assign()` возвращает значение типа `any`.

Листинг 4.19. Десериализация объекта Cat

```
class Cat {
    meow() {
        /* ... */
    }
}

let serializedCat: string = JSON.stringify(new Cat());

let deserializedCat: Cat =
    <Cat>Object.assign(new Cat(), JSON.parse(serializedCat));

deserializedCat.meow();
```

Десериализуем объект с помощью метода JSON.parse(), присваиваем его новому экземпляру типа Cat и приводим к типу Cat

Теперь можно вызвать для нашего объекта метод meow(), поскольку он приведен к типу Cat и имеет метод meow()

В ряде случаев при получении и десериализации большого количества возможных типов данных имеет смысл закодировать в сериализованный объект и некую информацию о типе. Например, описать протокол, в котором ко всем объектам спереди присоединяется символ, отражающий их тип. При этом можно кодировать объект `Cat`, добавив в начало полученной строки символ `"c"`. При получении сериа-

[1] Хочется верить, что при написании этого кода ни одна кошка не пострадала. — *Примеч. пер.*

лизованного объекта мы анализируем первый символ. Если это `"c"`, то можно безопасно приводить объект обратно к типу `Cat`. Если же этим символом окажется `"d"` (для типа `Dog`), то мы будем знать, что десериализовать к типу `Cat` данный объект нельзя, как показано в листинге 4.20.

Листинг 4.20. Сериализация с отслеживанием типа

```
class Cat {
    meow() {/* ... */ }
}

class Dog {
    bark() {/* ... */ }
}

function serializeCat(cat: Cat): string {
    return "c" + JSON.stringify(cat);
}

function serializeDog(dog: Dog): string {
    return "d" + JSON.stringify(dog);
}

function tryDeserializeCat(from: string): Cat | undefined {
    if (from[0] != "c") return undefined;

    return <Cat>Object.assign(new Cat(), JSON.parse(from.substr(1)));
}
```

Сериализуем объект Cat, добавляя в начало JSON-представления символ "c"

Сериализуем объект Dog, добавляя в начало JSON-представления символ "d"

Получив сериализованный объект, представляющий собой Cat или Dog, мы можем попытаться десериализовать Cat

Если первый символ не "c", то возвращаем undefined, поскольку десериализация в Cat невозможна

В противном случае применяем к оставшейся части строки функцию JSON.parse() и присваиваем результат ее выполнения объекту Cat

Сериализовав объект `Cat` и вызвав для его сериализованного представления метод `tryDeserializeCat()`, мы получим в ответ объект `Cat`. С другой стороны, сериализовав объект `Dog` и вызвав метод `tryDeserializeCat()`, мы получим в ответ `undefined`. Далее можно проверить, не получили ли мы `undefined`, и узнать, представляет ли наш объект собой `Cat`, как показано в листинге 4.21.

И хотя мы не могли ранее сравнить `Triangle` с `TLeft`, мы сравниваем `maybeCat` с `undefined`. Дело в том, что `undefined` — специальный единичный тип в TypeScript, у которого есть только одно вероятное значение — `undefined`. В отсутствие подобного типа всегда можно использовать тип вроде `Optional<Cat>`. Я рассказывал в главе 3, что `Optional<T>` — это тип, содержащий значение типа `T` или ничего.

Как мы видели на протяжении всей этой главы, типы делают возможными обеспечение безопасности кода на совершенно новом уровне. Допущения, которые раньше были неявными, теперь можно отражать в объявлениях типов и делать явными, избегая одержимости простыми типами данных и позволяя модулю проверки типов выявлять вероятные случаи неправильной интерпретации значений. Можно еще больше ограничить диапазон значений определенного типа

и обеспечить соблюдение ограничений при создании экземпляра. Это позволяет всегда гарантировать, что имеющийся экземпляр определенного типа — допустимый.

Листинг 4.21. Десериализация с отслеживанием типа

```
let catString: string = serializeCat(new Cat());
let dogString: string = serializeDog(new Dog());

let maybeCat: Cat | undefined = tryDeserializeCat(catString);

if (maybeCat != undefined) {
    let cat: Cat = <Cat>maybeCat;
    cat.meow();
}

maybeCat = tryDeserializeCat(dogString);
```

Сериализуем в строки объекты Cat и Dog

В результате вызова метода tryDeserializeCat возвращается Cat или undefined

Если да, то можем привести полученное к типу Cat и в результате получить объект, для которого можно вызвать метод meow()

Попытка десериализации сериализованного объекта Dog в объект Cat приведет к возврату undefined

Проверяем, получили ли мы Cat

С другой стороны, в некоторых ситуациях требуется бóльшая гибкость и желательно обрабатывать несколько типов схожим образом. В подобных случаях можно скрыть часть информации о типе и расширить множество значений, которые может принимать переменная. В большинстве случаев все равно желательно отслеживать первоначальный тип значения, чтобы иметь возможность восстановить его позднее. Мы делаем это вне системы типов, сохраняя информацию о типе где-то еще, например в другой переменной. А как только эта дополнительная гибкость становится не нужна и мы хотели бы снова положиться на модуль проверки типов, можно восстановить исходный тип с помощью приведения типов.

4.4.3. Упражнения

1. Какой тип необходимо использовать, чтобы можно было присвоить ему произвольное значение?
   А. `any`.
   Б. `unknown`.
   В. `any | unknown`.
   Г. Либо `any`, либо `unknown`.
2. Каково оптимальное представление для массива чисел и строк?
   А. `(number | string)[]`.
   Б. `number[] | string[]`.
   В. `unknown[]`.
   Г. `any[]`.

Резюме

- Антипаттерн одержимости простыми типами данных проявляется, когда мы объявляем значения базовых типов и делаем неявные допущения относительно их смысла.
- Альтернатива одержимости простыми типами данных — описание типов, явно отражающих смысл значений, что позволяет предотвратить их неправильное истолкование.
- Если необходимо наложить дополнительные ограничения, но нельзя сделать это на этапе компиляции, то можно обеспечить их соблюдение в конструкторах и фабриках и получить уверенность в корректности имеющегося объекта соответствующего типа.
- Иногда мы знаем больше, чем модуль проверки типов, поскольку можем хранить информацию о типах вне самой системы типов, в виде данных.
- Эту информацию можно использовать для выполнения безопасных приведений типов за счет предоставления модулю проверки типов дополнительной информации.
- Иногда может понадобиться обрабатывать различные типы одинаково, например, чтобы хранить значения различных типов в одной коллекции или их сериализации.
- Можно скрыть информацию, приведя значение к типу, включающему наш; к типу, который наследует наш тип; к типу-сумме или к типу, который может хранить значения любого другого типа.

До сих пор мы рассматривали базовые типы данных, способы их сочетания и другие способы использования систем типов для повышения безопасности кода. В главе 5 нас ждет нечто совершенно иное: мы обсудим, какие новые возможности открываются, если можно назначать функциям типы и работать с функциями так же, как и с любыми другими значениями в коде.

Ответы к упражнениям

4.1. Избегаем одержимости простыми типами данных, чтобы исключить неправильное толкование значений

В — наиболее безопасный способ — описать единицы измерения.

4.2. Обеспечиваем соблюдение ограничений

Вот одно из возможных решений:

```
declare const perc2entageType: unique symbol;

class Percentage {
    readonly value: number;
    [percentageType]: void;
```

```
    private constructor(value: number) {
        this.value = value;
    }

    static makePercentage(value: number): Percentage {
        if (value < 0) value = 0;
        if (value > 100) value = 100;

        return new Percentage(value);
    }
}
```

4.3. Добавляем информацию о типе

1. А — повышающее приведение типов безопасно (приведение дочернего типа к родительскому).
2. Б — понижающее приведение типов небезопасно (возможна потеря информации).

4.4. Скрываем и восстанавливаем информацию о типе

1. Б — `unknown` более безопасный вариант, чем `any`.
2. А — `unknown` и `any` уничтожают слишком много информации о типах.

5 Функциональные типы данных

В этой главе

- Упрощаем реализацию паттерна проектирования «Стратегия» с помощью функциональных типов данных.
- Реализация конечного автомата без операторов `switch`.
- Реализация отложенных значений в виде лямбда-выражений.
- Использование основополагающих алгоритмов обработки данных `map`, `filter` и `reduce` для снижения дублирования кода.

Мы рассмотрели основные типы данных и построенные на их основе типы. Кроме того, поговорили о том, как повысить безопасность программ с помощью объявления новых типов данных и обеспечить соблюдение разнообразных ограничений, накладываемых на их значения. Это практически все, чего можно добиться, используя алгебраические типы данных и комбинирование типов в типы-суммы и типы-произведения.

Следующая возможность систем типов, о которой мы поговорим, открывающая качественно новый уровень выражения логики, — типизация функций. Возможность именования функциональных типов данных и использования функций подобно значениям других типов (в качестве переменных, аргументов и возвращаемых типов данных функций) позволяет упростить реализацию нескольких распространенных языковых конструкций и вынести часто встречающиеся алгоритмы в библиотечные функции.

В этой главе мы рассмотрим способы упрощения реализации паттерна проектирования «Стратегия» (я также напомню вкратце, что он собой представляет, на случай, если вы забыли). Далее поговорим о конечных автоматах и более компактной их реализации с помощью функциональных свойств. Мы рассмотрим отложенные значения — возможность отсрочить дорогостоящие вычисления в надежде, что они нам не понадобятся. Наконец, обсудим основополагающие алгоритмы `map`, `filter` и `reduce`.

Все эти приложения возможны благодаря функциональным типам данных — следующей (за простейшими типами и их сочетаниями) ступеньке эволюции систем типов. А поскольку такие типы данных сегодня поддерживает большинство языков программирования, мы взглянем заново на несколько старых, испытанных и проверенных концепций.

5.1. Простой паттерн «Стратегия»

Один из чаще всего используемых паттернов проектирования — «Стратегия». Это поведенческий паттерн проектирования, позволяющий на этапе выполнения выбирать один алгоритм из семейства. Он расцепляет алгоритмы с использующими их компонентами, повышая таким образом гибкость системы в целом. Обычная архитектура этого паттерна изображена на рис. 5.1.

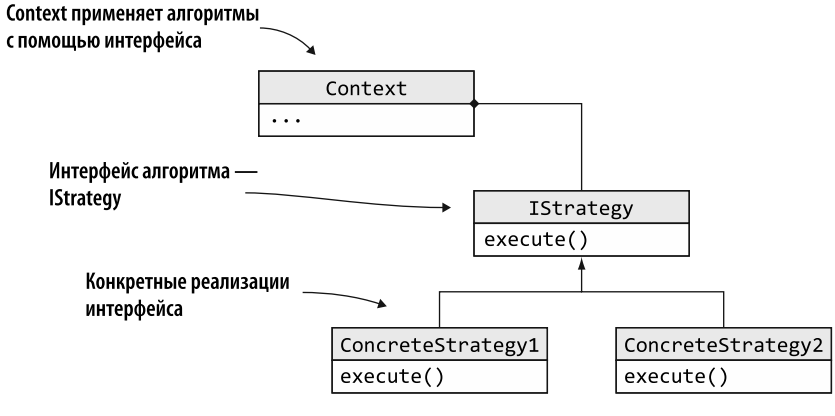


Рис. 5.1. Паттерн проектирования «Стратегия», состоящий из интерфейса IStrategy, реализаций ConcreteStrategy1 и ConcreteStrategy2, а также Context, применяющего алгоритмы с помощью интерфейса IStrategy

Рассмотрим конкретный пример. Пускай наша автомойка предоставляет две услуги: стандартную мойку и мойку премиум-класса (с дополнительной полировкой за три доллара).

Этот пример можно реализовать в виде стратегии (листинг 5.1), в которой интерфейс `IWashingStrategy` предоставляет метод `wash()`. Далее мы создадим две реализации этого интерфейса: `StandardWash` и `PremiumWash`. Класс `CarWash` пред-

ставляет собой контекст, применяющий метод `IWashingStrategy.wash()` к машине в зависимости от того, какую услугу оплатил пользователь.

Листинг 5.1. Стратегия для автомойки

```
class Car {
    /* представляет машину */
}

interface IWashingStrategy {
    wash(car: Car): void;
}

class StandardWash implements IWashingStrategy {
    wash(car: Car): void {
        /* проводит стандартную мойку */
    }
}

class PremiumWash implements IWashingStrategy {
    wash(car: Car): void {
        /* проводит мойку премиум-класса */
    }
}

class CarWash {
    service(car: Car, premium: boolean) {
        let washingStrategy: IWashingStrategy;

        if (premium) {
            washingStrategy = new PremiumWash();
        }else {
            washingStrategy = new StandardWash();
        }

        washingStrategy.wash(car);
    }
}
```

Класс Car представляет требующую мойки машину

IWashingStrategy — интерфейс стратегии, в котором объявлен метод wash()

StandardWash и PremiumWash — конкретные реализации этой стратегии

В зависимости от флага выбирается используемый алгоритм, после чего к экземпляру машины применяется метод wash()

Этот код вполне работоспособен, но слишком длинный. Он включает интерфейс и два реализующих типа, каждый из которых содержит один метод `wash()`. Эти типы на самом деле неважны; главное в коде — логика мойки машин. Данный код представляет собой всего лишь функцию, так что его можно существенно упростить, перейдя от интерфейсов и классов к функциональному типу данных и двум конкретным реализациям.

5.1.1. Функциональная стратегия

Опишем `WashingStrategy` — тип, представляющий собой функцию, которая получает в качестве аргумента `Car` и возвращает `void`. Далее реализуем два типа моек в виде двух функций: `standardWash()` и `premiumWash()`, получающих в качестве аргумента

Car и возвращающих void (листинг 5.2). Класс CarWash выбирает одну из них для применения к заданной машине.

Листинг 5.2. Переработанная стратегия для автомойки

```
class Car {
    /* представляет машину */
}

type WashingStrategy = (car: Car) => void;

function standardWash(car: Car): void {
     /* проводит стандартную мойку */
}

function premiumWash(car: Car): void {
     /* проводит мойку премиум-класса */
}

class CarWash {
    service(car: Car, premium: boolean) {
        let washingStrategy: WashingStrategy;

        if (premium) {
            washingStrategy = premiumWash;
        }else {
            washingStrategy = standardWash;
        }

        washingStrategy(car);
    }
}
```

WashingStrategy — функция, получающая в качестве аргумента Car и возвращающая void

Функции standardWash() и premiumWash() реализуют логику мойки машин

Теперь при выборе стратегии можно присвоить функцию непосредственно переменной washingStrategy

А поскольку переменная washingStrategy представляет собой функцию, можно просто ее вызвать

Эта реализация состоит из меньшего числа частей, чем предыдущая, как можно видеть на рис. 5.2.

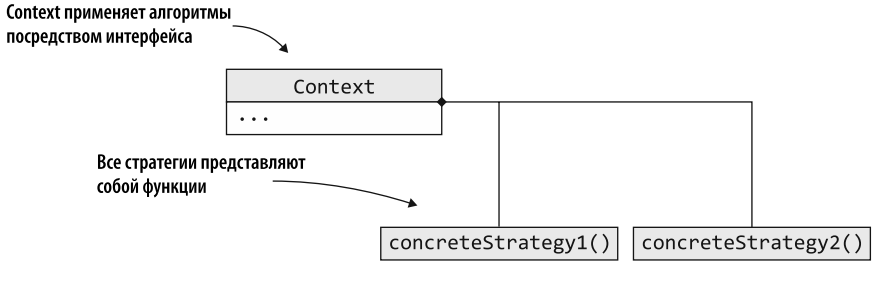


Рис. 5.2. Паттерн «Стратегия», состоящий из Context, применяющего одну из функций: либо concreteStrategy1(), либо concreteStrategy2()

Обсудим подробнее объявление функционального типа данных, поскольку мы сталкиваемся с ним впервые.

5.1.2. Типизация функций

Функция `standardWash()` получает в качестве аргумента `Car` и возвращает `void`, так что ее типом является *функция из Car в void* или в синтаксисе TypeScript: `(car: Car) => void`. Тип аргумента и возвращаемый тип функции `premiumWash()` — точно такие же, несмотря на отличающуюся реализацию, поэтому тип у нее тот же.

ФУНКЦИОНАЛЬНЫЙ ТИП (СИГНАТУРА)

Тип функции определяется типами ее аргументов и возвращаемым типом. Если у двух функций одинаковые аргументы и они возвращают значения одного типа, то у них один тип. Набор аргументов вместе с возвращаемым типом называется также сигнатурой функции.

Нам нужно ссылаться на этот тип, поэтому мы сделали его поименованным, с помощью объявления `type WashingStrategy = (car: Car) => void`. Используя `WashingStrategy` в качестве типа, мы подразумеваем функциональный тип `(car: Car) => void`. Мы ссылаемся на него в методе `CarWash.service()`.

А раз мы можем типизировать функции, значит, можем использовать представляющие функции переменные. В нашем примере переменная `washingStrategy` отражает функцию с только что приведенной сигнатурой. Мы можем присвоить этой переменной любую функцию, которая получает `Car` и возвращает `void`. Кроме того, мы можем вызывать ее как обычную функцию. В первом примере, где применялся интерфейс `IWashingStrategy`, наша логика мойки машин выполнялась с помощью вызова `washingStrategy.wash(car)`. Во втором же примере, где `washingStrategy` представляла собой функцию, мы просто вызвали `washingStrategy(car)`.

ПОЛНОПРАВНЫЕ ФУНКЦИИ

Возможность присваивать функции переменные и работать с ними как с любыми другими значениями системы типов приводит к так называемым полноправным функциям (first-class functions). Это значит, что данный язык программирования рассматривает функции как «полноправных граждан», предоставляя им те же права, что и другим значениям: у них есть тип, их можно присваивать переменным и передавать в качестве аргументов, проверять на допустимость и преобразовывать (в случае совместимости) в другие типы.

5.1.3. Реализации паттерна «Стратегия»

Ранее мы рассмотрели два способа реализации паттерна «Стратегия». Сравнивая эти две реализации, мы видим, что реализация стратегии «по всем правилам» из первого примера требует много дополнительных деталей: необходимо объявить интерфейс и иметь несколько реализующих его классов для конкретной логики данной стратегии. Вторая реализация сжата до самой сути того, что нам требуется: две реализующие нужную логику функции, на которые можно ссылаться непосредственно.

Обе реализации преследуют одну цель. Первая из них, основанная на интерфейсах, распространена больше, поскольку в эпоху популярности паттернов проектирования в 1990-е годы далеко не все, а скорее очень немногие из основных языков программирования поддерживали полноправные функции. Теперь все изменилось. Типизация функций доступна в большинстве языков, и мы можем воспользоваться этим, чтобы создавать более компактные реализации некоторых паттернов.

Важно учитывать, что *паттерн* не меняется: мы по-прежнему инкапсулируем семейство алгоритмов и выбираем один из них на этапе выполнения. Отличие лишь в реализации, которую современные возможности языков позволяют выражать намного проще. Мы заменяем интерфейс и два класса (каждый из них реализует метод) на объявление типа и две функции.

В большинстве случаев такой более лаконичной реализации вполне достаточно. Реализация с интерфейсом и классом может понадобиться, когда алгоритмы не получается представить в виде простых функций. Иногда требуется несколько функций или нужно отслеживать какое-либо состояние. В таком случае более уместна первая из наших реализаций, поскольку группирует связанные части стратегии в общий тип данных.

5.1.4. Полноправные функции

Прежде чем продолжить, вкратце напомню основные понятия, с которыми вы познакомились в этом разделе.

- Набор аргументов вместе с возвращаемым функцией значением называется *сигнатурой* функции. У следующих двух функций — одинаковые сигнатуры:

```
function add(x: number, y: number): number {
    return x + y;
}

function subtract(x: number, y: number): number {
    return x - y;
}
```

- Сигнатура функции эквивалентна ее *типу* в языках, где можно типизировать функции. Тип предыдущих двух функций: *функция из (number, number) в number* или `(x: number, y: number) => number`. Обратите внимание: фактические названия аргументов неважны: тип `(a: number, b: number) => number` идентичен типу `(x: number, y: number) => number`.
- Если язык программирования позволяет работать с функциями точно так же, как с любыми другими значениями, то говорят, что он поддерживает *полноправные функции*. Функции можно присваивать переменным, передавать в качестве аргументов и использовать аналогично любым другим значениям, что значительно повышает выразительность кода.

5.1.5. Упражнения

1. Каков тип функции `isEven()`, принимающей в качестве аргумента число и возвращающей `true`, если число четное, и `false` в противном случае?
   А. `[number, boolean]`.
   Б. `(x: number) => boolean`.
   В. `(x: number, isEven: boolean)`.
   Г. `{x: number, isEven: boolean}`.
2. Каков тип функции `check()`, принимающей число и функцию того же типа, что `isEven()`, в качестве аргументов и возвращающей результат применения этой функции к данному значению?
   А. `(x: number, func: number) => boolean`.
   Б. `(x: number) => (x: number) => boolean`.
   В. `(x: number, func: (x: number) => boolean) => boolean`.
   Г. `(x: number, func: (x: number) => boolean) => void`.

5.2. Конечные автоматы без операторов switch

Одно из очень удобных приложений полноправных функций — возможность описания свойства класса с функциональным типом данных. Это позволяет присваивать ему различные функции, меняя поведение во время выполнения. Фактически получается подключаемый метод класса, который можно менять при необходимости.

Так, можно реализовать подключаемый класс `Greeter` (от англ. to greet — «приветствовать, здороваться») (листинг 5.3). Вместо реализации метода `greet()` мы реализуем свойство `greet` с функциональным типом данных. Далее мы сможем присваивать ему функции с различными приветствиями, например `sayGoodMorning()` (пожелать доброго утра) и `sayGoodNight()` (пожелать спокойной ночи).

Листинг 5.3. Подключаемый Greeter

```
function sayGoodMorning(): void {
    console.log("Good morning!");
}

function sayGoodNight(): void {
    console.log("Good night!");
}

class Greeter {
    greet: () => void = sayGoodMorning;
}

let greeter: Greeter = new Greeter();

greeter.greet();
```

Две функции, выводящие приветствия в консоль

greet — функция без аргументов, возвращающая void, по умолчанию принимающая значение sayGoodMorning()

Поскольку greet — функциональное свойство, можно вызывать его точно так же, как и любой метод класса

```
greeter.greet = sayGoodNight;
```

Ему можно присвоить другую функцию

```
greeter.greet();
```

В результате второго вызова будет вызвана функция sayGoodNight()

Все это логично следует из реализации паттерна «Стратегия», обсуждавшейся в предыдущем разделе. Однако стоит отметить: данный подход позволяет легко добавить в класс подключаемое поведение. Чтобы добавить новое приветствие, достаточно просто добавить еще одну функцию с той же сигнатурой и присвоить ее свойству `greet`.

5.2.1. Предварительная версия книги

Работая над рукописью, я написал маленький сценарий для синхронизации исходного кода с текстом книги. Данный набросок я писал на популярном языке разметки Markdown. Я хранил исходный код в отдельных файлах TypeScript, чтобы иметь возможность компилировать их и проверять работоспособность после обновления примеров.

Мне нужно было обеспечить актуальность примеров кода в тексте Markdown. Они всегда располагаются между строкой, содержащей ```ts, и строкой, содержащей ```. При генерации HTML на основе исходного кода в формате Markdown ```ts интерпретируется как начало блока кода TypeScript, визуализируемого с выделением синтаксических элементов TypeScript, а ``` отмечает конец этого блока кода. Содержимое этих блоков должно было вставляться из файлов исходного кода на TypeScript, которые можно скомпилировать и проверить вне текста (рис. 5.3).

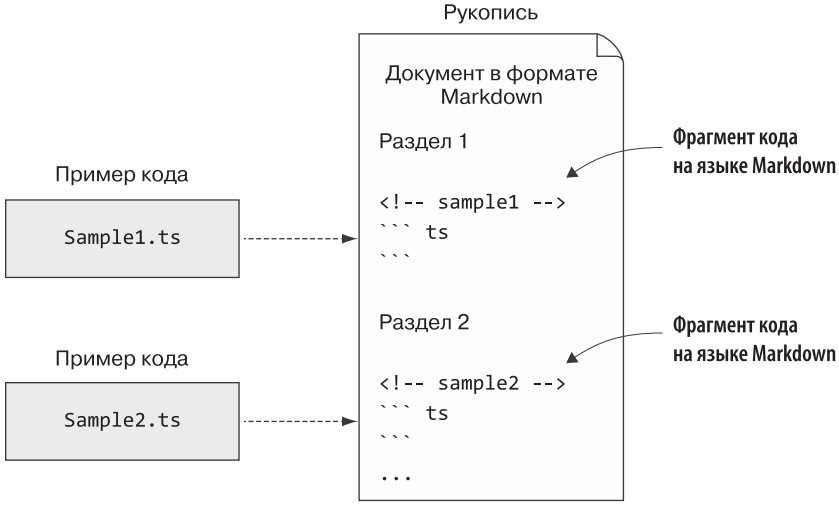


Рис. 5.3. Два файла TypeScript (.ts), содержащих примеры кода, встраиваемые в документы в формате Markdown между маркерами ```ts и ```. Примеры для моего сценария снабжены комментарием <!-- ... -->

Чтобы выяснить, куда должен попасть тот или иной пример, я использовал небольшой трюк. Язык Markdown допускает применение чистого HTML в тексте документа, поэтому я снабдил каждый из примеров кода HTML-комментарием наподобие `<!-- sample1 -->`. HTML-комментарии не визуализируются, так что после преобразования Markdown в HTML они оказываются невидимыми. С другой стороны, мой сценарий может использовать эти комментарии с целью выяснить, куда следует встроить тот или иной пример кода.

При загрузке всех этих примеров кода с диска мне приходилось обрабатывать все Markdown-документы и создавать обновленную версию следующим образом.

- В режиме обработки текста просто копировать каждую из строк входного текста в выходной документ в неизменном виде. А натолкнувшись на маркер (`<!-- sample -->`), извлекать соответствующий пример кода и переключаться в режим обработки маркеров.
- В режиме обработки маркеров снова копировать все строки входного текста в выходной документ, пока не встретится маркер блока кода (`` ```ts ``). А встретив маркер кода, выводить актуальную версию примера кода, загруженную из файла TypeScript, и переключаться в режим обработки кода.
- В режиме обработки кода мы уже обеспечили попадание в выходной документ последней версии кода, поэтому можем пропустить, вероятно, устаревшую версию из блока кода. Пропускаем все строки, пока не встретим маркер конца блока кода (` ``` `). Далее переключаемся обратно в режим обработки текста.

При каждом запуске существующие примеры кода в документе, перед которыми указан маркер `<!-- ... -->`, обновляются в соответствии с актуальной версией из TypeScript-файлов на диске. Другие блоки кода, без предшествующего маркера `<!-- ... -->`, не обновляются, поскольку обрабатываются в режиме обработки текста.

Вот, скажем, пример кода `helloWorld.ts` (листинг 5.4).

Листинг 5.4. helloWorld.ts

```
console.log("Hello world!");
```

Мы хотели бы встроить этот код в `Chapter1.md`, причем гарантировать поддержание его актуальности, как показано в листинге 5.5.

Листинг 5.5. Chapter1.md

```
# Chapter 1

Printing "Hello world!".
<!-- helloWorld -->
```ts
console.log("Hello");
```
```

Не совсем актуальный код. Строка здесь гласит "Hello", что не соответствует файлу helloWorld.ts

Этот документ обрабатывается построчно следующим образом.

1. В режиме обработки текста `"Chapter 1"` копируется в выходной документ в неизменном виде.
2. `""` (пустая строка) копируется в выходной документ в неизменном виде.

3. `"Printing "Hello world!"."` копируется в выходной документ в неизменном виде. Впрочем, это маркер, так что мы фиксируем пример кода, который необходимо вставить (**`helloWorld.ts`**), и переключаемся в режим обработки маркера.
4. `"```ts"` копируется в выходной документ в неизменном виде. Это маркер блока кода, так что сразу после копирования его в выходной документ мы также выводим туда содержимое **`helloWorld.ts`**. Кроме того, переключаемся в режим обработки кода.
5. Строку `"console.log("Hello");"` мы пропускаем. Мы не копируем строки в режиме обработки кода, поскольку меняем их на свежую версию из файла примера кода.
6. `"```"` представляет собой маркер конца блока кода. Мы вставляем его, после чего переключаемся обратно в режим обработки текста.

5.2.2. Конечные автоматы

Удобнее всего моделировать поведение нашего сценария обработки текста в виде конечного автомата. У такого автомата есть два набора: состояний и переходов между парами состояний. Автомат начинает работу с заданного состояния, называемого также *начальным* (start state), и при соблюдении определенных условий переходит в другое состояние.

Именно это и происходит с нашим обработчиком текста и его тремя режимами обработки. Входные строки обрабатываются определенным образом в *режиме обработки текста*. А при соблюдении некоего условия (при обнаружении маркера `<!-- sample -->`) наш обработчик переходит в *режим обработки маркера*. И снова при определенном другом условии (обнаружении маркера блока кода `` ```ts ``) переходит в *режим обработки кода*. А встречая маркер конца блока кода (`` ``` ``), возвращается в *режим обработки текста* (рис. 5.4).

Рис. 5.4. Конечный автомат обработки текста с тремя состояниями (обработка текста, маркера, кода) и переходами между ними в зависимости от входных данных. Начальное состояние — обработка текста

Теперь, смоделировав наше решение, мы можем обсудить его доступные реализации. Один из способов реализации конечного автомата — описание набора состояний в виде перечисляемого типа данных, отслеживание текущего состояния и достижение необходимого поведения с помощью оператора `switch`, охватывающего все возможные состояния. В нашем случае можно описать перечисляемый тип `TextProcessingMode`.

Класс `TextProcessor` будет хранить текущее состояние в свойстве `mode` и реализовывать оператор `switch` в методе `processLine()`. В зависимости от состояния этот метод будет по очереди вызывать один из трех методов обработки: `processTextLine()`, `processMarkerLine()` и `processCodeLine()`. В этих функциях мы реализуем обработку текста, а затем (в соответствующем случае) переход в другое состояние путем обновления текущего состояния.

Обработка документа в формате Markdown, состоящего из многих строк текста, означает обработку всех строк по очереди с помощью нашего конечного автомата и возврат конечного результата вызывающей стороне, как показано в листинге 5.6.

Листинг 5.6. Реализация конечного автомата

```
enum TextProcessingMode {
    Text,
    Marker,
    Code,
}

class TextProcessor {
    private mode: TextProcessingMode = TextProcessingMode.Text;
    private result: string[] = [];
    private codeSample: string[] = [];

    processText(lines: string[]): string[] {
        this.result = [];
        this.mode = TextProcessingMode.Text;

        for (let line of lines) {
            this.processLine(line);
        }

        return this.result;
    }

    private processLine(line: string): void {
        switch (this.mode) {
            case TextProcessingMode.Text:
                this.processTextLine(line);
                break;
            case TextProcessingMode.Marker:
                this.processMarkerLine(line);
                break;
            case TextProcessingMode.Code:
                this.processCodeLine(line);
                break;
        }
    }
```

Состояния отражены в перечисляемом типе данных

Обработка текстового документа: обработка всех строк и возврат получившегося в итоге массива строк

Оператор switch конечного автомата вызывает соответствующий обработчик в зависимости от текущего состояния

```
    private processTextLine(line: string): void {
        this.result.push(line);

        if (line.startsWith("<!--")) {
            this.loadCodeSample(line);

            this.mode = TextProcessingMode.Marker;
        }
    }

    private processMarkerLine(line: string): void {
        this.result.push(line);

        if (line.startsWith("```ts")) {
            this.result = this.result.concat(this.codeSample);

            this.mode = TextProcessingMode.Code;
        }
    }

    private processCodeLine(line: string): void {
        if (line.startsWith("```")) {
            this.result.push(line);

            this.mode = TextProcessingMode.Text;
        }
    }

    private loadCodeSample(line: string) {
        /* загружаем пример кода в зависимости от маркера
           и сохраняем его в this.codeSample */
    }
}
```

Обработка строки текста. Если строка начинается с "<!--", то загружаем пример кода и переходим в следующее состояние

Обработка маркера. Если строка начинается с "```ts", то вставляем пример кода и переходим в следующее состояние

Обработка кода с пропуском строк. Если строка начинается с "```", то переходим в состояние (режим) обработки текста

Тело этой функции мы опустили, поскольку для данного примера оно неважно

Мы опустили код загрузки примера кода из внешнего файла, поскольку он не особенно важен для нашего обсуждения конечных автоматов. Эта реализация вполне работоспособна, но ее можно упростить, воспользовавшись подключаемой функцией.

Обратите внимание: сигнатура всех наших функций обработки текста одинакова: они принимают на входе строку текста в виде аргумента типа `string` и возвращают `void`. Что, если вместо реализации в `processLine()` большого оператора `switch` с переходом к соответствующей функции мы сделаем `processLine()` одной из этих функций?

Вместо реализации `processLine()` в виде метода мы можем описать ее в виде свойства класса с типом `(line: string) => void` и начальным значением `processTextLine()`, как показано в листинге 5.7. Далее в каждом из трех методов обработки текста вместо установки различных значений `mode` из перечисляемого типа мы будем устанавливать значение `processLine` равным различным методам. Фактически нам больше не нужно отслеживать состояние во внешней переменной. Нам даже больше не требуется перечисляемый тип!

Листинг 5.7. Другой вариант реализации конечного автомата

```
class TextProcessor {
    private result: string[] = [];
    private processLine: (line: string) => void = this.processTextLine;
    private codeSample: string[] = [];

    processText(lines: string[]): string[] {
        this.result = [];
        this.processLine = this.processTextLine;

        for (let line of lines) {
            this.processLine(line);
        }

        return this.result;
    }

    private processTextLine(line: string): void {
        this.result.push(line);

        if (line.startsWith("<!--")) {
            this.loadCodeSample(line);

            this.processLine = this.processMarkerLine;
        }
    }

    private processMarkerLine(line: string): void {
        this.result.push(line);

        if (line.startsWith("```ts")) {
            this.result = this.result.concat(this.codeSample);

            this.processLine = this.processCodeLine;
        }
    }

    private processCodeLine(line: string): void {
        if (line.startsWith("```")) {
            this.result.push(line);

            this.processLine = this.processTextLine;
        }
    }

    private loadCodeSample(line: string) {
        /* загружаем пример кода в зависимости от маркера
           и сохраняем его в this.codeSample */
    }
}
```

Переходы из состояния в состояние теперь осуществляются путем замены значения свойства this.processLine на соответствующий метод

В этой второй реализации мы избавились от перечисляемого типа `TextProcessingMode`, свойства `mode` и оператора `switch`, который делегировал обработку

соответствующему методу. Вместо того чтобы делегировать обработку, свойство `processLine` теперь само является ее соответствующим методом.

Для этой реализации не нужно отслеживать состояния по отдельности и согласовывать их с логикой обработки. Если требуется ввести в автомат новое состояние, то в старой реализации пришлось бы модифицировать код в нескольких местах. Помимо реализации новой логики обработки и переходов из состояния в состояние, пришлось бы обновить перечисляемый тип и добавить еще один пункт в оператор `switch`. Во второй же реализации этого не требуется: состояние представлено только функцией.

Конечные автоматы на основе типов-сумм

В случае конечных автоматов с большим количеством состояний захват состояний или даже переходов между ними явным образом позволил бы сделать код более понятным. Но даже несмотря на это, вместо перечисляемых типов данных и операторов `switch` можно создать реализацию, в которой все состояния были бы представлены в виде отдельных типов, а весь конечный автомат — в виде типа-суммы возможных состояний. Это позволило бы разбить архитектуру на типобезопасные компоненты. Ниже приведен пример реализации конечного автомата на основе типа-суммы. Код несколько «раздут», поэтому по возможности лучше использовать обсуждавшуюся выше реализацию как еще одну альтернативу конечному автомату на основе `switch`.

При использовании типа-суммы для каждого состояния применяется отдельный тип, в данном случае `TextLineProcessor`, `MarkerLineProcessor` и `CodeLineProcessor`. Каждый из них ведет учет обработанных на текущий момент строк в члене класса `result` и включает метод `process()`, осуществляющий обработку строки текста.

Конечный автомат на основе типа-суммы

```
class TextLineProcessor {
    result: string[];

    constructor(result: string[]) {
        this.result = result;
    }

    process(line: string): TextLineProcessor | MarkerLineProcessor {
        this.result.push(line);

        if (line.startsWith("<!--")) {
            return new MarkerLineProcessor(
                this.result, this.loadCodeSample(line));
        } else {
            return this;
        }
    }

    private loadCodeSample(line: string): string[] {
        /* загружаем пример кода в зависимости от маркера
           и сохраняем его в this.codeSample */
    }
}
```

TextLineProcessor возвращает либо TextLineProcessor, либо MarkerLineProcessor для обработки следующей строки

Если строка начинается с "<!--", то возвращаем новый объект MarkerLineProcessor; в противном случае возвращаем текущий обработчик (this)

```
class MarkerLineProcessor {
    result: string[];
    codeSample: string[]

    constructor(result: string[], codeSample: string[]) {
        this.result = result;
        this.codeSample = codeSample;
    }

    process(line: string): MarkerLineProcessor | CodeLineProcessor {
        this.result.push(line);

        if (line.startsWith("```ts")) {
            this.result = this.result.concat(this.codeSample);

            return new CodeLineProcessor(this.result);
        } else {
            return this;
        }
    }
}

class CodeLineProcessor {
    result: string[];

    constructor(result: string[]) {
        this.result = result;
    }

    process(line: string): CodeLineProcessor | TextLineProcessor {
        if (line.startsWith("```")) {
            this.result.push(line);

            return new TextLineProcessor(this.result);
        } else {
            return this;
        }
    }
}

function processText(lines: string): string[] {
    let processor: TextLineProcessor | MarkerLineProcessor
        | CodeLineProcessor = new TextLineProcessor([]);

    for (let line of lines) {
        processor = processor.process(line);
    }

    return processor.result;
}
```

MarkerLineProcessor возвращает либо MarkerLineProcessor, либо CodeLineProcessor

Если встречаем "```ts", то загружаем пример кода и возвращаем новый объект CodeLineProcessor; в противном случае возвращаем текущий обработчик (this)

CodeLineProcessor возвращает либо CodeLineProcessor, либо TextLineProcessor

Если строка начинается с "```", то добавляем ее в конец результата и возвращаем новый объект TextLineProcessor; иначе возвращаем текущий обработчик (this)

Состояния представлены объектом processor — типом-суммой типов TextLineProcessor, MarkerLineProcessor и CodeLineProcessor

processor обновляется после каждой обработанной строки в случае изменения состояния

Все наши обработчики возвращают экземпляр обработчика `this`, если состояние не изменилось, или в противном случае новый обработчик. Функция `processText()` выполняет конечный автомат, вызывая `process()` для каждой строки текста и обновляя поле `processor` при изменении состояния, присваивая ему результат вызова метода.

Теперь набор состояний отражен явным образом в сигнатурах переменной `processor`, которая может быть `TextLineProcessor`, `MarkerLineProcessor` или `CodeLineProcessor`.

Возможные переходы отражаются в сигнатурах методов `process()`. Например, `TextLineProcessor.process` возвращает `TextLineProcessor | MarkerLineProcessor`, так что может либо остаться в том же состоянии (`TextLineProcessor`), либо перейти в состояние `MarkerLineProcessor`. У этих классов состояний при необходимости могут быть другие свойства и члены классов. Данная реализация несколько длиннее реализации на основе функций, так что если эти дополнительные возможности не нужны, то лучше использовать более простое решение.

5.2.3. Краткое резюме по реализации конечного автомата

Вкратце резюмируем обсуждавшиеся в этом разделе различные реализации, после чего перейдем к другим приложениям функциональных типов данных.

- В «традиционной» реализации конечного автомата используется перечисляемый тип данных для описания всех возможных состояний, переменная этого типа для хранения текущего состояния и большой оператор `switch` для выбора нужного вида обработки в зависимости от текущего состояния. Переходы между состояниями реализованы путем обновления переменной текущего состояния. Недостаток этой реализации — разделение состояний и производимой во время каждого из них обработки, вследствие чего компилятор не может предотвратить случаи выполнения обработки, не соответствующей состоянию. Ничто не мешает нам, например, вызывать `processCodeLine()`, находясь в состоянии `TextProcessingMode.Text`. Кроме того, приходится хранить состояние и переходы в отдельной переменной перечисляемого типа, рискуя потерять согласованность (например, мы можем добавить в перечисляемый тип данных новое значение, но забыть добавить вариант для него в оператор `switch`).
- При функциональной реализации каждый режим обработки представляет собой функцию, а для отслеживания текущего состояния используется функциональное свойство. Переходы между состояниями реализованы с помощью присваивания другого состояния функциональному свойству. Это достаточно облегченная реализация, подходящая для многих сценариев применения. У нее, впрочем, есть два недостатка: иногда необходимо связать с каждым из состояний больше информации и хотелось бы описывать возможные состояния и переходы между ними явным образом.
- В реализации на основе типа-суммы для всех состояний обработки используются отдельные классы, а отслеживание текущего состояния выполняется с помощью переменной типа-суммы всех возможных состояний. Переходы между состоя-

ниями реализованы путем присваивания другого состояния этой переменной, благодаря чему можно добавлять свойства и члены в состояния и группировать их. Недостаток этого подхода — больший объем кода, чем у функциональной реализации.

На этом наше обсуждение конечных автоматов завершается. В следующем разделе мы рассмотрим еще один способ применения функциональных типов данных: реализацию отложенных вычислений.

5.2.4. Упражнения

1. Смоделируйте в виде конечного автомата простое соединение, которое может быть открыто (`open`) или закрыто (`closed`). Для открытия соединения используется метод `connect`, а закрытия — `disconnect`.
2. Реализуйте предыдущее соединение в виде функционального конечного автомата с функцией `process()`. В случае закрытого соединения функция `process()` должна его открыть. В случае открытого соединения — вызвать функцию `read()`, которая возвращает строку. Если та пуста, то соединение должно закрываться; в противном случае необходимо вывести в консоль возвращенную функцией `read()` строку. Функция `read()` должна быть объявлена в виде `declare function read(): string;`.

5.3. Избегаем ресурсоемких вычислений с помощью отложенных значений

Еще одно преимущество использования функций как обычных значений — возможность их хранения и вызова в случае надобности. Иногда вычисление необходимого значения бывает весьма ресурсоемким. Допустим, наша программа может создавать объекты `Bike` (Велосипед) и `Car` (Автомобиль). Например, нам нужен объект `Car`, но его создание является очень ресурсоемким, так что вместо него мы поедем на велосипеде. Создание объекта `Bike` требует очень мало ресурсов, поэтому затраты на него нас не волнуют. Вместо того чтобы создавать объект `Car` при каждом запуске программы для применения его при необходимости, не лучше ли создавать `Car` по запросу? В этом случае можно запрашивать создание объекта `Car`, когда это действительно нужно, и только тогда выполнять ресурсоемкую логику его создания. Если мы никогда не запросим его создание, то никакие ресурсы не будут потрачены впустую.

Идея заключается в следующем: отложить ресурсоемкие вычисления на максимально более поздний срок в надежде, что они не потребуются. А поскольку они выражаются в виде функций, можно передавать функции вместо фактических значений и вызывать их тогда и в том случае, если эти значения понадобятся. Данный процесс

носит название *отложенного вычисления* (lazy evaluation). Его противоположность — *немедленное вычисление* (eager evaluation), при котором значения генерируются и передаются сразу же, даже если потом могут не понадобиться (листинг 5.8).

Листинг 5.8. Немедленное формирование объекта Car

```
class Bike {}
class Car {}

function chooseMyRide(bike: Bike, car: Car): Bike | Car {
    if (isItRaining()) {
        return car;
    } else {
        return bike;
    }
}

chooseMyRide(new Bike(), new Car());
```

Классы Car и Bike. Допустим, создание экземпляра Car требует много ресурсов

Функция chooseMyRide() выбирает Bike или Car в зависимости от некоего условия

Для вызова функции chooseMyRide () необходимо создать объект Car

В нашем примере с немедленным созданием `Car` для вызова функции `chooseMyRide` необходимо передать в нее `Car`, поэтому мы сразу же тратим ресурсы на формирование объекта `Car`. И если погода окажется отличной и я решу поехать на велосипеде, то получится, что объект `Car` был создан впустую.

Перейдем к отложенному подходу. Вместо того чтобы передавать `Car`, мы передадим функцию, возвращающую при вызове объект `Car` (листинг 5.9).

Листинг 5.9. Отложенное формирование объекта Car

```
class Bike {}
class Car {}

function chooseMyRide(bike: Bike, car: () => Car): Bike | Car {
    if (isItRaining()) {
        return car();
    } else {
        return bike;
    }
}

function makeCar(): Car {
    return new Car();
}

chooseMyRide(new Bike(), makeCar);
```

Вместо аргумента типа Car функция chooseMyRide() теперь принимает в качестве параметра функцию, возвращающую объект Car

Эта функция вызывается, только когда нам действительно нужен объект Car

Обертываем процесс создания машины в функцию и передаем ее в chooseMyRide()

В этой отложенной версии дорогостоящий объект `Car` создается, только если действительно нужен. Реши я поехать на велосипеде, функция вообще не будет вызвана и объект `Car` не создастся.

Это можно реализовать и с помощью чисто объектно-ориентированных конструкций, хотя кода потребуется намного больше. Можно объявить класс `CarFactory` в качестве обертки для метода `makeCar()` и воспользоваться им в качестве аргумента

функции `chooseMyRide()`. А затем создавать новый экземпляр `CarFactory` при вызове `chooseMyRide()`, вызывая упомянутый метод при необходимости. Но зачем писать больше кода, если можно обойтись меньшим объемом? На самом деле наш код можно сократить еще больше.

5.3.1. Лямбда-выражения

Большинство современных языков программирования поддерживает *анонимные функции*, или *лямбда-выражения*. Они напоминают обычные функции, только без названий. Лямбда-выражения применяются в контексте, где требуются «одноразовые» функции: такие, к которым мы собираемся обратиться лишь раз, поэтому давать ей название — только делать лишнюю работу. Вместо этого лучше использовать встраиваемую реализацию.

В нашем примере с отложенным созданием автомобиля хороший кандидат на роль такого лямбда-выражения — метод `makeCar()`. Поскольку для функции `chooseMyRide()` необходима функция без аргументов, возвращающая `Car`, нам нужно объявить новую функцию, на которую мы ссылаемся только один раз: передавая ее в качестве аргумента в `chooseMyRide()`. Вместо этой функции можно использовать анонимную, как показано в листинге 5.10.

Листинг 5.10. Создание объекта Car с помощью анонимной функции

```
class Bike {}
class Car {}

function chooseMyRide(bike: Bike, car: () => Car): Bike | Car {
    if (isItRaining()) {
        return car();
    } else {
        return bike;
    }
}

chooseMyRide(new Bike(), () => new Car());
```

Лямбда-выражение без аргументов, возвращающее объект Car

Синтаксис лямбда-выражений TypeScript очень напоминает объявление функциональных типов данных: в скобках указывается список аргументов (в данном случае их нет), далее символ `=>`, а затем тело функции. Если функция состоит из нескольких строк, то они помещаются между `{` и `}`. Но в данном случае мы выполняем только один вызов `new Car()`, который неявно рассматривается как оператор возврата лямбда-выражения, поэтому избавляемся от `makeCar()` и можем поместить логику создания экземпляра в однострочную функцию.

ЛЯМБДА-ВЫРАЖЕНИЕ (АНОНИМНАЯ ФУНКЦИЯ)

Лямбда-выражение (анонимная функция) — это описание функции без названия. Лямбда-выражения обычно используются для одноразовой, краткой обработки и передаются аналогично обычным данным.

Лямбда-выражения бесполезны, если нет возможности типизации функций. Что можно сделать с таким выражением, как `() => new Car()`? Если нельзя сохранить его в переменной или передать в качестве аргумента в другую функцию, то и пользы от него немного. С другой стороны, возможность передачи функций подобно обычным значениям позволяет реализовать сценарии, аналогичные вышеприведенному, в котором код отложенного создания объекта `Car` лишь на несколько символов длиннее версии с немедленным его созданием.

Отложенные вычисления

Распространенная возможность многих функциональных языков программирования — *отложенные вычисления*. В подобных языках все вычисляется как можно позднее, и не обязательно указывать это явно. В таких языках функция `chooseMyRide()` не создавала бы по умолчанию ни `Bike`, ни `Car`. Любой из этих объектов был бы создан, только когда мы действительно попытались бы воспользоваться возвращаемым `chooseMyRide()` объектом — например, вызвав его метод `ride()`.

Императивные языки программирования, такие как TypeScript, Java, C# и C++, ориентированы на *немедленное вычисление*. Тем не менее и в них, как мы видели выше, при необходимости можно легко смоделировать отложенные вычисления. Мы рассмотрим дополнительные примеры этого, когда будем обсуждать генераторы.

5.3.2. Упражнение

В каком из следующих фрагментов кода реализовано лямбда-выражение, складывающее два числа?

А. `function add(x: number, y: number)=> number { return x + y; }`.

Б. `add(x: number, y: number) => number { return x + y; }`.

В. `add(x: number, y: number) { return x + y; }`.

Г. `(x: number, y: number) => x + y;`.

5.4. Использование операций map, filter и reduce

Рассмотрим еще одну возможность, возникающую благодаря типизации функций: функции, принимающие другие функции как аргументы или возвращающие их. «Обычную» функцию, которая принимает один или несколько нефункциональных аргументов и возвращает нефункциональный тип данных, называют также *функцией первого порядка* (first-order function), это простая, самая обыкновенная функция. Функции же, принимающие функции первого порядка в качестве аргументов или возвращающие функции первого порядка, называют *функциями второго порядка* (second-order function).

Можно взобраться еще выше по этой лестнице и сказать, что функция, принимающая функции второго порядка в качестве аргументов или возвращающая функцию второго порядка, называется *функцией третьего порядка* (third-order function). Однако на практике все функции, принимающие или возвращающие другие функции, называют *функциями высшего порядка* (higher-order functions).

Примером функции высшего порядка может послужить вторая версия функции `chooseMyRide()` из предыдущего раздела. Этой функции требуется аргумент типа `() => Car`, который сам является функцией.

Фактически оказывается, что в виде функций высшего порядка можно реализовать несколько очень полезных алгоритмов, основные из которых — `map()`, `filter()` и `reduce()`. В большинство языков программирования включены библиотеки, содержащие версии этих функций, но мы создадим их реализации своими руками и изучим все подробности.

5.4.1. Операция map()

Основная идея операции `map()` очень проста: вызвать функцию для каждого из значений определенного типа, содержащихся в заданной коллекции, и вернуть коллекцию результатов этих вызовов. Подобная разновидность обработки встречается на практике регулярно, так что имеет смысл сократить возможное дублирование кода.

Рассмотрим в качестве примеров два сценария. Во-первых, возведение в квадрат каждого из числовых значений в заданном массиве. Во-вторых, вычисление длины каждого из строковых значений в заданном массиве.

Эти примеры можно реализовать с помощью пары циклов `for`. Но если взглянуть на них рядом друг с другом, то возникает ощущение, что часть их общих черт можно выделить в некий совместно используемый код (листинг 5.11).

Листинг 5.11. Специализированные алгоритмы отображения

```
let numbers: number[] = [1, 2, 3, 4, 5];
let squares: number[] = [];

for (const n of numbers) {
    squares.push(n * n);
}

let strings: string[] = ["apple", "orange", "peach"];
let lengths: number[] = [];

for (const s of strings) {
    lengths.push(s.length);
}
```

Массив чисел

Возводим в квадрат каждое из чисел в массиве и вставляем результат в массив squares

Массив строк

Вычисляем длину каждой из строк в массиве и вставляем результат в массив lengths

Хотя массивы и преобразования различаются, схемы алгоритмов очень похожи (рис. 5.5).

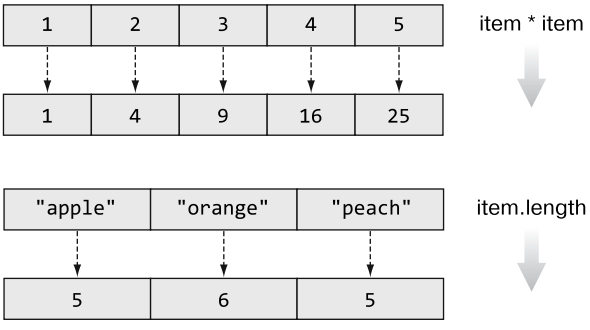


Рис. 5.5. Возведение чисел в квадрат и получение длин строк — очень разные сценарии, однако общая схема преобразования одна: берем входной массив, применяем функцию и генерируем выходной массив

Реализация отображения своими руками

Посмотрим на реализацию `map()` для массивов и подумаем, как можно избежать многократного написания одинаковых циклов. Воспользуемся обобщенными типами `T` и `U`, поскольку реализация работает одинаково, вне зависимости от того, каковы данные типы. Таким образом мы сможем применить эту функцию для различных типов данных, а не будем ограничивать ее, скажем, массивами чисел.

Наша функция принимает на входе массив значений типа `T` и функцию, принимающую элемент типа `T` в качестве аргумента и возвращающую значение типа `U`. Результат собирается в массив значений типа `U`. Реализация в листинге 5.12 просто обходит все элементы массива значений типа `T`, применяя к каждому из них заданную функцию, после чего сохраняет результат в массиве значений типа `U`.

Листинг 5.12. Операция map()

```
function map<T, U>(items: T[], func: (item: T) => U): U[] {
    let result: U[] = [];

    for (const item of items) {
        result.push(func(item));
    }

    return result;
}
```

Операция map() принимает на входе массив элементов типа T и функцию, переводящую из T в U, и возвращает массив значений типа U

Вначале массив значений типа U пуст

Для каждого результата вставляем результат выполнения func(item) в массив значений типа U

Возвращаем массив значений типа U

В этой простой функции инкапсулирован общий код обработки из предыдущего примера. Благодаря операции `map()` можно сгенерировать массив квадратов и массив длин строк с помощью пары однострочных операторов, как показано в листинге 5.13.

Листинг 5.13. Использование операции map()

```
let numbers: number[] = [1, 2, 3, 4, 5];
let squares: number[] = map(numbers, (item) => item * item);

let strings: string[] = ["apple", "orange", "peach"];
let lengths: number[] = map(strings, (item) => item.length);
```

Вызываем операцию map() с помощью лямбда-выражения (item) => item * item (в данном случае item — число)

Вызываем операцию map() с помощью лямбда-выражения (item) => item.length (в данном случае item — символьная строка)

Функция `map()` инкапсулирует применение функции, передаваемой ей в качестве аргумента. Можно просто передать ей массив элементов и функцию, и она вернет массив, полученный в результате использования этой функции. Далее, когда мы будем обсуждать обобщенные типы данных, вы увидите, как можно обобщить эту функцию для работы с произвольной структурой данных, а не только с массивами. Впрочем, даже с текущей реализацией получилась отличная абстракция применения функций к наборам элементов, которую можно использовать во множестве сценариев.

5.4.2. Операция filter()

Следующий весьма распространенный сценарий, двоюродный брат `map()`, — `filter()`: фильтрация заданной коллекции элементов по принципу соответствия заданному условию и возврат коллекции соответствующих ему элементов.

Вернемся к нашим примерам с числами и строками и отфильтруем список, оставив в нем только четные числа и строки длины 5. Функция `map()` тут не поможет, поскольку обрабатывает все элементы в коллекции, а мы в данном случае хотим отбросить некоторые из них. Специализированная реализация опять же включала бы проход в цикле по коллекции и проверку соответствия условию, как показано в листинге 5.14.

Листинг 5.14. Специализированная реализация фильтрации

```
let numbers: number[] = [1, 2, 3, 4, 5];
let evens: number[] = []

for (const n of numbers) {
    if (n % 2 == 0) {
        evens.push(n);
    }
}

let strings: string[] = ["apple", "orange", "peach"];
let length5Strings: string[] = [];

for (const s of strings) {
    if (s.length == 5) {
        length5Strings.push(s);
    }
}
```

Помещаем элемент, только если он четный

Помещаем элемент, только если его длина равна 5

И вновь сразу заметна общая для обеих реализаций структура (рис. 5.6).

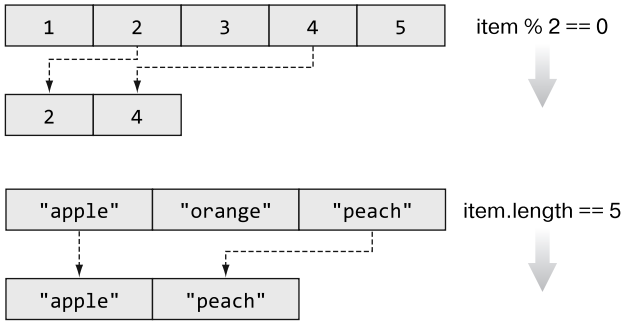


Рис. 5.6. Общая структура вычисления четных чисел и строк длиной 5. Производится обход входных данных, применение фильтра и возврат элементов, для которых фильтр возвращает true

Фильтр своими руками

Аналогично проделанному с `map()` мы можем реализовать обобщенную функцию высшего порядка `filter()`, принимающую в качестве аргументов массив входных данных и функцию-фильтр и возвращающую отфильтрованные результаты, как показано в листинге 5.15. В данном случае при входном массиве типа `T` функция-фильтр — это функция, которая принимает в качестве аргументов `T` и возвращает `boolean`. Функцию, принимающую на входе один аргумент и возвращающую `boolean`, называют *предикатом* (predicate).

Листинг 5.15. filter()

```
function filter<T>(items: T[], pred: (item: T) => boolean): T[] {
    let result: T[] = [];

    for (const item of items) {
        if (pred(item)) {
            result.push(item);
        }
    }

    return result;
}
```

Функция filter() принимает в качестве аргументов массив значений типа T и предикат (функцию из T в boolean)

Если предикат возвращает true, то добавляем элемент в итоговый массив, в противном случае пропускаем его

Посмотрим, как выглядит код фильтрации при использовании общей структуры, реализованной в нашей функции `filter()`. Как четные числа, так и строки длиной `5` вычисляются за одну строку кода в листинге 5.16.

Листинг 5.16. Использование filter()

```
let numbers: number[] = [1, 2, 3, 4, 5];
let evens: number[] = filter(numbers, (item) => item % 2 == 0);

let strings: string[] = ["apple", "orange", "peach"];
let length5Strings: string[] = filter(strings, (item) => item.length == 5);
```

Фильтрация массивов производится на основе предиката. В первом случае это лямбда-выражение, возвращающее `true`, если число делится на 2. А во втором случае — лямбда-выражение, возвращающее `true`, если длина строки равна `5`.

Мы реализовали нашу вторую распространенную операцию в виде обобщенной функции. Теперь перейдем к третьей, последней из операций, которые хотели рассмотреть в этой главе.

5.4.3. Операция reduce()

Пока что мы научились применять функцию к коллекции элементов с помощью операции `map()` и удалять элементы, не соответствующие определенному критерию, с помощью операции `filter()`. Третья часто встречающаяся операция объединяет все элементы коллекции в одно значение.

Например, нам может понадобиться вычислить произведение всех чисел в массиве или произвести конкатенацию всех строк в массиве в одну большую строку. Эти сценарии различаются, однако обладают общей базовой структурой. Для начала рассмотрим специализированную реализацию (листинг 5.17).

Листинг 5.17. Специализированная операция свертки

```
let numbers: number[] = [1, 2, 3, 4, 5];
let product: number = 1;

for (const n of numbers) {
    product = product * n;
}

let strings: string[] = ["apple", "orange", "peach"];
let longString: string = "";

for (const s of strings) {
    longString = longString + s;
}
```

В случае произведения начинаем с начального значения 1

Умножаем product на каждое из чисел в нашей коллекции, накапливая результат

В случае строк начинаем с пустой строки

Присоединяем строки по одной к пустой строке, накапливая результат

В обоих случаях мы начинаем с начального значения, а затем накапливаем результат, проходя по коллекции и группируя каждый из элементов со значением-накопителем. По завершении обхода коллекций `product` содержит произведение всех чисел из массива `numbers`, а `longString` представляет собой конкатенацию всех строк из массива `strings` (рис. 5.7).

Свертка своими руками

В листинге 5.18 мы реализовали обобщенную функцию, принимающую массив элементов типа `T`, его начальное значение и функцию, принимающую два аргумента типа `T` и возвращающую `T`. Промежуточный итог мы будем хранить в локальной переменной и обновлять его, применяя вышеупомянутую функцию к ней и к каждому из элементов входного массива по очереди.

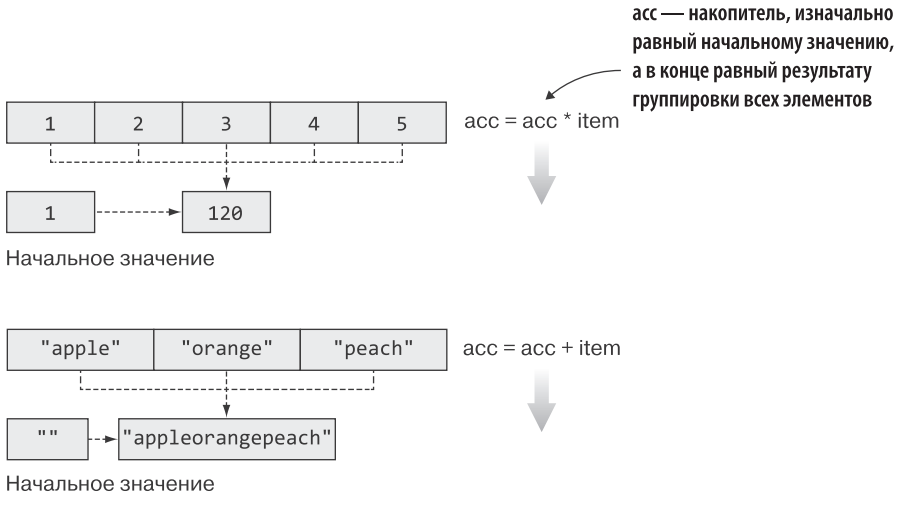


Рис. 5.7. Общая структура группировки чисел из числового массива и строк из массива строк. В первом случае начальное значение равно 1, а операция группировки представляет собой умножение на каждый из элементов. Во втором случае начальное значение равно "", а операция группировки представляет собой конкатенацию с каждым из элементов

Листинг 5.18. Операция reduce()

```
function reduce<T>(items: T[], init: T, op: (x: T, y: T) => T): T {
    let result: T = init;

    for (const item of items) {
        result = op(result, item);
    }

    return result;
}
```

Функция reduce() принимает в качестве аргументов массив значений типа T, начальное значение и операцию группировки двух значений типа T в одно

Все элементы массива группируются с промежуточным итогом с помощью заданной операции

У этой функции три аргумента, а у двух предыдущих — по два. Нам приходится использовать начальное значение, а не начинать, скажем, с первого элемента массива, поскольку массив может оказаться пустым. Если в коллекции нет ни одного элемента, то чему должен быть равен `result`? В подобной ситуации можно просто вернуть начальное значение, для этого оно и нужно.

Теперь взглянем, как можно модифицировать наши специализированные реализации для использования `reduce()` (листинг 5.19).

У операции `reduce()` есть несколько нюансов, отсутствующих у двух других. Помимо того что требуется начальное значение, на итоговый результат может влиять порядок группировки элементов. Это не относится к операциям и начальным значениям из нашего примера. Но если бы начальной строкой была, скажем, `"banana"`? Тогда при конкатенации слева направо в результате получилось бы `"bananaappleorangepeach"`. А при обходе массива справа налево и добавлении элементов в начало строки мы получили бы `"appleorangepeachbanana"`.

Или, например, применение операции группировки, состоящей в объединении первых символов строк, сначала к `"apple"` и `"orange"` дает `"ao"`. Применение ее далее

к "ao" и "peach" дает "ap". С другой стороны, если начать с "orange" и "peach", то получается "op". А затем из "apple" и "op" получается "ao" (рис. 5.8).

Листинг 5.19. Использование reduce()

```
let numbers: number[] = [1, 2, 3, 4, 5];
let product: number = reduce(numbers, 1, (x, y) => x * y);

let strings: string[] = ["apple", "orange", "peach"];
let longString: string = reduce(strings, "", (x, y) => x + y);
```

Для чисел начинаем с начального значения 1 и используем операцию (x, y) => x * y (умножение)

Для строк начинаем с начального значения "" и используем операцию (x, y) => x + y (конкатенация)

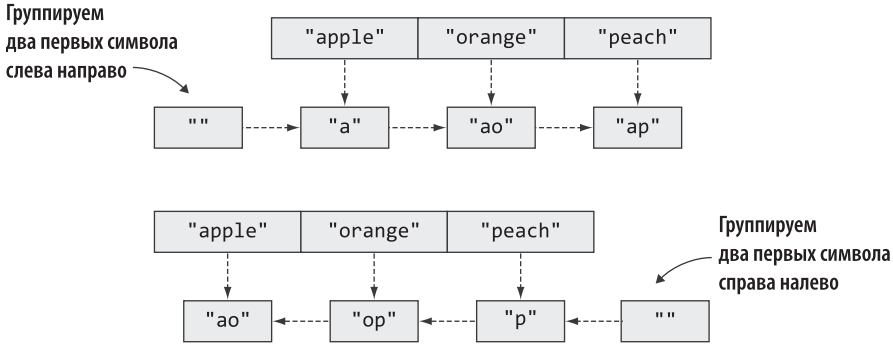


Рис. 5.8. Группировка массива строк с помощью операции «первые буквы обеих строк» дает различные результаты в случае применения слева направо и справа налево. В первом случае мы начинаем с пустой строки и "apple", получаем "a" и "orange", далее "ao" и "peach" и в результате "ap". Во втором начинаем с пустой строки и "peach", за которыми следуют "orange" и "p", что дает "op", и, наконец, "apple" и "op", что дает "ao"

Традиционно операция `reduce()` применяется слева направо, поэтому можете смело считать, что любая встреченная вами ее библиотечная реализация работает именно так. В некоторых библиотеках имеется и версия, работающая справа налево. Например, в типе `Array` языка TypeScript есть как метод `reduce()`, так и метод `reduceRight()`. Если вас интересует математический аппарат, лежащий в основе этой операции, то смотрите врезку «Моноиды».

Моноиды

Абстрактная алгебра оперирует множествами и операциями над ними. Как мы уже видели ранее, тип можно рассматривать в качестве множества его вероятных значений. Операцию над типом `T`, принимающую на входе два объекта `T` и возвращающую другой объект типа `T`, `(T, T) => T`, можно рассматривать как операцию над множеством значений данного типа. Например, множество для типа `number` и операция `+`, то есть `(x, y) => x + y`, образуют алгебраическую структуру.

Подобные структуры определяются свойствами своих операций. *Единичный элемент* (identity) — это элемент `id` типа `T`, для которого `op(x, id) == op(id, x) == x`. Другими словами, группировка `id` с любым другим элементом оставляет этот другой элемент неизменным. Единичным элементом является `0` в случае множества `number` и операции сложения; `1` — в случае множества `number` и операции умножения и `""` (пустая строка) — в случае множества `string` и операции конкатенации строк.

Операция называется ассоциативной, если порядок применения ее к последовательности элементов неважен, то есть конечный результат все равно не изменится. Для любых значений `x`, `y`, `z` типа `T` — `op(x, op(y, z)) == op(op(x, y), z)`. Это равенство соблюдается, например, для сложения и умножения чисел, в отличие от вычитания и нашей операции «первые символы обеих строк».

Если у множества `T` с определенной на нем операцией `op` существует единичный элемент и эта операция ассоциативна, то полученная в итоге алгебраическая структура называется *моноидом* (monoid). В случае моноида свертка, начинаемая с единичного элемента как начального значения, слева направо и справа налево дает одинаковый результат. Можно даже убрать требование относительно начального значения и единичного значения в качестве умолчания при пустой коллекции. Кроме того, свертку можно распараллелить. Произвести свертку первой и второй половин коллекции параллельно, а затем объединить результаты, поскольку свойство ассоциативности гарантирует получение того же результата. В случае массива `[1, 2, 3, 4, 5, 6]` можно сгруппировать `1 + 2 + 3` и параллельно `4 + 5 + 6`, а затем сложить результаты.

Но если отказаться от одного из вышеупомянутых свойств, то гарантии теряются. Без ассоциативности, при наличии просто множества, операции и единичного элемента, хоть начального значения и не требуется (мы воспользуемся единичным элементом), начинает играть роль направление применения операций. Без единичного элемента, но с ассоциативностью получается *полугруппа*. При отсутствии единичного элемента важно, где мы помещаем начальное значение: слева от первого элемента или справа от последнего.

Основной вывод из вышеизложенного: операция `reduce()` прекрасно работает для моноидов. Если же речь идет не о моноиде, то следует обратить внимание на используемое начальное значение и направление свертки.

5.4.4. Библиотечная поддержка

Как уже упоминалось в начале данного раздела, большинство языков программирования поддерживают эти распространенные алгоритмы на уровне библиотек. Впрочем, эти алгоритмы могут встречаться под самыми разными названиями, поскольку не существует единого стандарта их наименования.

В C# операции `map()`, `filter()` и `reduce()` можно найти в пространстве имен `System.Linq` под названиями `Select()`, `Where()` и `Aggregate()` соответственно. В Java они включены в пакет `java.util.stream` и называются `map()`, `filter()` и `reduce()`.

Операция `map()` может называться также `Select()` или `transform()`. Операция `filter()` может называться `Where()`. Операция `reduce()` может носить название `accumulate()`, `Aggregate()` или `fold()`, в зависимости от языка и библиотеки.

Однако, несмотря на многообразие названий, эти алгоритмы являются основополагающими и используются в самых разнообразных приложениях. В дальнейшем мы обсудим многие подобные алгоритмы, но именно эти три формируют фундамент обработки данных с помощью функций высшего порядка.

Знаменитый фреймворк MapReduce компании Google, предназначенный для крупномасштабной обработки данных, применяет те же базовые принципы `map()` и `reduce()` путем выполнения массово-параллельной операции `map()` на множестве узлов и объединения результатов с помощью `reduce()`-подобной операции.

5.4.5. Упражнения

1. Реализуйте функцию `first()`. Она должна принимать массив значений типа `T` и функцию `pred` (предикат), получающую в качестве аргумента значение типа `T` и возвращающую `boolean`. Функция `first()` должна возвращать первый элемент массива, для которого `pred()` вернет `true` или `undefined`, если `pred()` возвращает `false` для всех элементов.
2. Реализуйте функцию `all()`. Она должна принимать массив значений типа `T` и функцию `pred` (предикат), получающую в качестве аргумента значение типа `T` и возвращающую `boolean`. Функция `all()` должна возвращать `true`, если `pred()` возвращает `true` для всех элементов массива, и `false` в противном случае.

5.5. Функциональное программирование

Хоть рассмотренный в этой главе материал несколько сложнее представленного выше, есть и хорошая новость: мы обсудили большинство ключевых составляющих функционального программирования. Синтаксис некоторых функциональных языков может сбить с толку разработчиков, привыкших к императивным, объектно-ориентированным языкам. Их системы типов поддерживают типы-суммы, типы-произведения и функции первого порядка, а также множество библиотечных функций для обработки данных, таких как `map()`, `filter()` и `reduce()`. Во многих функциональных языках программирования применяется отложенное вычисление, которое также обсуждалось в этой главе.

Благодаря типизации функций становится возможной реализация многих идей функциональных языков программирования в нефункциональных (или не чисто функциональных) языках. В данной главе были затронуты все эти вопросы и показаны императивные реализации всех ключевых компонентов.

Резюме

- Благодаря типизации функций можно гораздо проще реализовать паттерн проектирования «Стратегия», сосредоточив внимание только на функциях, реализующих логику, и забыть об окружающем скаффолдинге.
- Благодаря возможности подключить функцию в класс в виде свойства и вызывать как метод можно реализовывать конечные автоматы без огромных

операторов `switch`. Таким образом, компилятор может предотвращать ошибки, например не позволять применить случайно ошибочный вид обработки в каком-то заданном состоянии.

- Типы-суммы, в которых каждому состоянию соответствует свой тип, — еще одна альтернатива операторам `switch`.
- Отложенные значения (функций-оберток для дорогостоящих вычислений) позволяют откладывать на потом вычисления, требующие больших затрат ресурсов. Их можно вызывать, при необходимости генерировать значение или не вызывать вовсе, пропуская затратные вычисления, если значение не понадобилось.
- Функция высшего порядка — функция, которая принимает другую функцию как аргумент или возвращает ее.
- Три основные функции высшего порядка, широко применяемые для обработки данных, — `map()`, `filter()` и `reduce()`.

В главе 6 мы рассмотрим еще несколько приложений типизированных функций. Расскажем о замыканиях и упрощении с их помощью еще одного распространенного паттерна проектирования — паттерна «Декоратор». Кроме того, обсудим промисы, а также выполнение заданий и событийно-управляемые системы. Все эти приложения стали возможны благодаря представлению вычислений (функции) в виде «полноправных граждан» системы типов.

Ответы к упражнениям

5.1. Простой паттерн «Стратегия»

1. Б — это единственный функциональный тип; остальные объявления не описывают функции.
2. В — функция принимает `number` и `(x: number) => boolean` и возвращает `boolean`.

5.2. Конечные автоматы без операторов switch

1. Требуемое соединение можно смоделировать в виде конечного автомата с двумя состояниями — `open` и `closed` — и двумя переходами из одного состояния в другое — `connect` для перехода из состояния `closed` в `open` и `disconnect` для перехода из `open` в `closed`.
2. Одна из возможных реализаций:

```
declare function read(): string;

class Connection {
    private doProcess: () => void = this.processClosedConnection;
    public process(): void {
        this.doProcess();
    }
```

```
    private processClosedConnection() {
        this.doProcess = this.processOpenConnection;
    }

    private processOpenConnection() {
        const value: string = read();

        if (value.length == 0) {
            this.doProcess = this.processClosedConnection;
        } else {
            console.log(value);
        }
    }
}
```

5.3. Избегаем ресурсоемких вычислений с помощью отложенных значений

Г — это единственная анонимная реализация; в остальных вариантах ответа реализованы поименованные функции.

5.4. Использование операций map, filter и reduce

1. Одна из возможных реализаций `first()`:

```
function first<T>(items: T[], pred: (item: T) => boolean):
    T | undefined {
    for (const item of items) {
        if (pred(item)) {
            return item;
        }
    }

    return undefined;
}
```

2. Одна из возможных реализаций `all()`:

```
function all<T>(items: T[], pred: (item: T) => boolean): boolean {
    for (const item of items) {
        if (!pred(item)) {
            return false;
        }
    }

    return true;
}
```

6 Расширенные возможности применения функциональных типов данных

В этой главе

- Использование упрощенного паттерна проектирования «Декоратор».
- Реализация возобновляемого счетчика.
- Обработка длительных операций.
- Написание понятного асинхронного кода с помощью промисов и конструкции `async/await`.

В главе 5 мы рассмотрели основы функциональных типов данных и сценарии, ставшие возможными благодаря работе с функциями подобно любым другим значениям, то есть передаче их в качестве аргументов и возврате в виде результатов. Мы также рассмотрели несколько весьма многообещающих абстракций, реализующих распространенные паттерны обработки данных: `map()`, `filter()` и `reduce()`.

В этой главе мы продолжим обсуждение функциональных типов данных и их более продвинутых приложений. Начнем с паттерна проектирования «Декоратор» и его реализаций — традиционной и альтернативной. (Повторю: не волнуйтесь, если подзабыли его; я напомню, что к чему.) Вы познакомитесь с понятием *«замыкание»* (closure) и узнаете, как с его помощью реализовать простой счетчик. Затем рассмотрим другой способ реализации счетчика, на этот раз задействовав генератор — функцию, выдающую несколько результатов.

Далее мы поговорим об асинхронных операциях. Рассмотрим две основные модели асинхронного выполнения кода: потоки выполнения и циклы ожидания событий — и узнаем, как распланировать выполнение нескольких длительных операций. Мы начнем с функций обратного вызова, затем рассмотрим промисы и, наконец, поговорим о синтаксисе `async/await`, доступном сегодня в большинстве основных языков программирования.

Как мы увидим на последующих страницах, все обсуждаемые в этой главе темы возможны лишь благодаря использованию функций в качестве значений.

6.1. Простой паттерн проектирования «Декоратор»

«Декоратор» — это поведенческий паттерн проектирования программного обеспечения, который расширяет поведение объекта, не прибегая к модификации соответствующего класса. Декорированный объект способен на выполнение задач, выходящих за рамки возможностей его исходной реализации. Схема этого паттерна приведена на рис. 6.1.

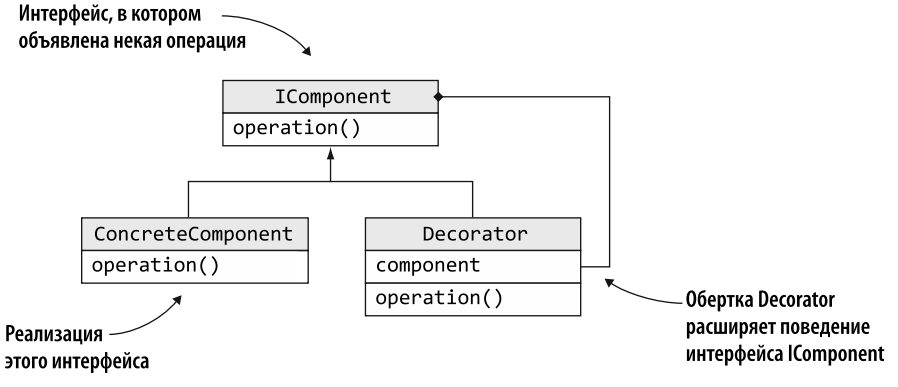


Рис. 6.1. Паттерн «Декоратор»: интерфейс IComponent, его конкретная реализация ConcreteComponent и Decorator, расширяющий IComponent дополнительным поведением

Для примера представим, что у нас есть интерфейс `IWidgetFactory`, в котором объявлен метод `Widget()`, возвращающий объект `Widget`. А в конкретной реализации `WidgetFactory` реализован метод для создания новых объектов `Widget`.

Допустим, что мы хотим повторно использовать `Widget` и вместо того, чтобы создавать каждый раз новый объект, хотели бы создать только один объект класса и всегда возвращать его (то есть реализовать одиночку). Не внося изменений в класс `WidgetFactory`, мы можем создать декоратор `SingletonDecorator` — обертку для `IWidgetFactory`, как показано в листинге 6.1, и расширить его поведение так, чтобы создавался лишь один объект `Widget` (рис. 6.2).

Листинг 6.1. Декоратор для IWidgetFactory

```
class Widget {}

interface IWidgetFactory {
    makeWidget(): Widget;
}

class WidgetFactory implements IWidgetFactory {
    public makeWidget(): Widget {
        return new Widget();
    }
}

class SingletonDecorator implements IWidgetFactory {
    private factory: IWidgetFactory;
    private instance: Widget | undefined = undefined;

    constructor(factory: IWidgetFactory) {
        this.factory = factory;
    }

    public makeWidget(): Widget {
        if (this.instance == undefined) {
            this.instance = this.factory.makeWidget();
        }

        return this.instance;
    }
}
```

WidgetFactory просто создает новый объект Widget

SingletonDecorator обертывает IWidgetFactory

Метод makeWidget реализует логику одиночки и гарантирует, что может быть создан только один экземпляр Widget

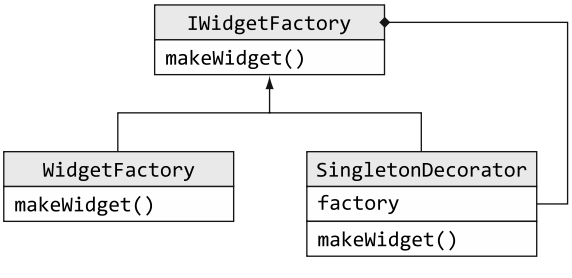


Рис. 6.2. Паттерн «Декоратор» для фабрики виджетов. IWidgetFactory — интерфейс, WidgetFactory — конкретная реализация, а класс SingletonDecorator добавляет в IWidgetFactory поведение одиночки

Преимущество этого паттерна заключается в поддержке *принципа единственной обязанности* (single-responsibility principle), который гласит: класс должен отвечать только за что-то одно. В данном случае класс `WidgetFactory` отвечает за создание виджетов, а `SingletonDecorator` — за поведение, соответствующее одиночке. Если нам потребуется несколько экземпляров класса, то можно воспользоваться непосредственно классом `WidgetFactory`. Если же один — классом `SingletonDecorator`.

6.1.1. Функциональный декоратор

Попробуем упростить эту реализацию опять-таки с помощью типизированных функций. Для начала избавимся от интерфейса `IWidgetFactory`, заменив его функциональным типом данных, описывающим функцию без аргументов, которая возвращает объект `Widget`: `() => Widget`.

Теперь мы можем заменить класс `WidgetFactory` простой функцией `makeWidget()`. Там, где раньше использовался интерфейс `IWidgetFactory` и передавался экземпляр `WidgetFactory`, теперь мы потребуем функции типа `() => Widget` и будем передавать туда `makeWidget()`, как показано в листинге 6.2.

Листинг 6.2. Функциональная фабрика виджетов

```
class Widget {}

type WidgetFactory = () => Widget;

function makeWidget(): Widget {
    return new Widget();
}

function use10Widgets(factory: WidgetFactory) {
    for (let i = 0; i < 10; i++) {
        let widget = factory();
        /* ... */
    }
}

use10Widgets(makeWidget);
```

Функциональный тип данных для фабрики виджетов

Тип функции makeWidget() соответствует типу WidgetFactory

Функция use10Widgets требует наличия параметра типа WidgetFactory и использует его для создания десяти экземпляров Widget

Пример вызова: передаем функцию makeWidget в качестве аргумента

Для создания функциональной фабрики виджетов мы используем методику, очень близкую к паттерну проектирования «Стратегия» из главы 5: передаем функцию в качестве аргумента и вызываем ее при необходимости. Теперь посмотрим, как добавить сюда поведение одиночки.

Создаем новую функцию, `singletonDecorator()`, принимающую в качестве аргумента функцию типа `WidgetFactory` и возвращающую другую функцию типа `WidgetFactory`. Как вы помните из главы 5, лямбда-выражение — это функция без названия, которую можно возвращать из другой функции. В листинге 6.3 наш декоратор получает фабрику и с ее помощью создает новую функцию, отвечающую за поведение одиночки (рис. 6.3).

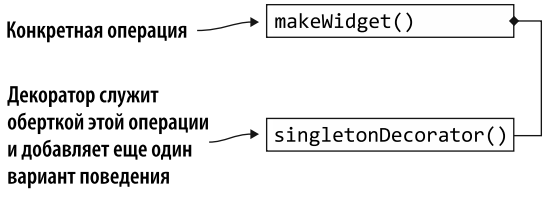


Рис. 6.3. Функциональный декоратор: теперь достаточно функций makeWidget() и singletonDecorator()

Листинг 6.3. Декоратор для функциональной фабрики виджетов

```
class Widget {}

type WidgetFactory = () => Widget;

function makeWidget(): Widget {
    return new Widget();
}

function singletonDecorator(factory: WidgetFactory): WidgetFactory {
    let instance: Widget | undefined = undefined;

    return (): Widget => {
        if (instance == undefined) {
            instance = factory();
        }
        return instance;
    };
}

function use10Widgets(factory: WidgetFactory) {
    for (let i = 0; i < 10; i++) {
        let widget = factory();
        /* ... */
    }
}

use10Widgets(singletonDecorator(makeWidget));
```

Функция singletonDecorator() возвращает лямбда-выражение, реализующее поведение одиночки, используя заданную фабрику для создания объекта Widget

А поскольку функция singletonDecorator() возвращает WidgetFactory, ее можно передать в качестве аргумента функции use10Widgets()

Теперь вместо создания десяти объектов `Widget` функция `use10Widgets()` вызывает лямбда-выражение, повторно использующее один и тот же объект `Widget` для всех вызовов.

В этом коде количество компонентов уменьшается с интерфейса и двух классов — по одному методу каждый (конкретная операция и декоратор) — до двух функций.

6.1.2. Реализации декоратора

Как и в случае нашего паттерна «Стратегия», объектно-ориентированный и функциональный подход реализуют один и тот же паттерн проектирования «Декоратор». Объектно-ориентированная версия требует объявления интерфейса (`IWidgetFactory`), по крайней мере одной реализации этого интерфейса (`WidgetFactory`) и класса-декоратора, отвечающего за дополнительный вариант поведения (`SingletonDecorator`). При функциональной реализации же, напротив, просто объявляется тип для фабричной функции (`() => Widget`) и используются две функции: функция-фабрика (`makeWidget()`) и функция-декоратор (`singletonDecorator()`).

Стоит отметить, что в функциональном случае тип декоратора отличается от типа `makeWidget()`. У фабрики аргументов нет, она возвращает `Widget`, а декоратор принимает на входе фабрику виджетов и возвращает другую. Говоря иначе,

`singletonDecorator()` принимает в качестве аргумента функцию и возвращает ее в качестве результата. Это возможно только благодаря полноправности функций, то есть возможности работать с функциями точно так же, как и с прочими переменными, и использовать их в качестве аргументов и возвращаемых значений.

Эта более компактная реализация, ставшая доступной благодаря современным системам типов, вполне подходит для многих случаев. Более «многословное» объектно-ориентированное решение подходит для работы с несколькими функциями. Если в нашем интерфейсе объявлено несколько методов, то заменить их одним функциональным типом данных не получится.

6.1.3. Замыкания

Посмотрим более внимательно на реализацию `singletonDecorator()` в листинге 6.4. Возможно, вы обратили внимание на интересный нюанс: хоть функция возвращает лямбда-выражение, оно ссылается как на аргумент `factory`, так и на, казалось бы, локальную (по отношению к функции `singletonDecorator()`) переменную `instance`.

Листинг 6.4. Функция-декоратор

```
function singletonDecorator(factory: WidgetFactory): WidgetFactory {
    let instance: Widget | undefined = undefined;

    return (): Widget => {
        if (instance == undefined) {
            instance = factory();
        }

        return instance;
    };
}
```

И даже после возврата из функции `singletonDecorator()` переменная `instance` все равно существует, поскольку была «захвачена» лямбда-выражением. Это явление называется *лямбда-захватом* (lambda capture).

ЗАМЫКАНИЯ И ЛЯМБДА-ЗАХВАТЫ

Лямбда-захват представляет собой захват внешней переменной внутри лямбда-выражения. Такие захваты реализуются в языках программирования с помощью замыканий. Замыкание — это не просто функция: оно также фиксирует среду, в которой функция была создана, так что может сохранять состояние от вызова до вызова.

В нашем случае переменная `instance` в функции `singletonDecorator()` является частью такой среды, поэтому возвращенное лямбда-выражение по-прежнему сможет ссылаться на `instance` (рис. 6.4).

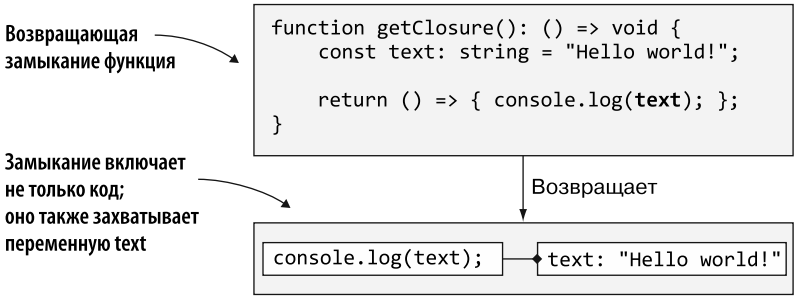


Рис. 6.4. Простая функция, возвращающая замыкание: лямбда-выражение, которое ссылается на локальную (по отношению к этой функции) переменную. Даже после возврата из функции getClosure() замыкание все равно ссылается на переменную, так что она существует дольше, чем функция, в которой появляется

Замыкания имеют смысл только при наличии функций высшего порядка. Если нельзя вернуть из одной функции другую, то нет и среды, которую можно было бы захватить. В этом случае все функции находятся в глобальной области видимости, которая и играет роль их среды. Они могут ссылаться на глобальные переменные.

Можно также сравнить замыкания с объектами. Объект — некое состояние с набором методов; *замыкание* — функция с неким захваченным состоянием. Рассмотрим еще один пример, в котором нам пригодятся замыкания, — реализацию счетчика.

6.1.4. Упражнение

Реализуйте функцию `loggingDecorator()`, принимающую в качестве аргумента другую функцию, `factory()`, которая не принимает аргументов и возвращает объект `Widget`. Декоратор должен вывести в консоль `"Widget created"`, прежде чем с помощью вызова заданной (переданной ему) функции вернуть объект `Widget`.

6.2. Реализация счетчика

Рассмотрим очень простой сценарий: создание счетчика, возвращающего последовательные числа, начиная с 1. Этот пример может показаться тривиальным, однако охватывает несколько возможных реализаций, которые можно применять любому сценарию генерации значений. Один из этих вариантов реализации — воспользоваться глобальной переменной и функцией, которая возвращает ее, после чего увеличивает ее значение на 1, как показано в листинге 6.5.

Данная реализация работает, но она не оптимальна. Во-первых, `n` — глобальная переменная, так что доступ к ней есть у кого угодно. Другой код может изменить ее значение незаметно для нас. Во-вторых, это реализация одного счетчика. А что, если нам понадобятся два счетчика, начинающихся с 1?

Листинг 6.5. Глобальный счетчик

```
let n: number = 1;

function next() {
    return n++;
}

console.log(next());
console.log(next());
console.log(next());
```

Счетчик хранится в глобальной переменной

Функция next() возвращает n, после чего увеличивает значение n на 1

В результате должно выводиться:
1
2
3

6.2.1. Объектно-ориентированный счетчик

Первая реализация, которую мы рассмотрим, — объектно-ориентированная, возможно, хорошо вам знакомая. Мы создадим класс `Counter`, в котором в качестве приватного члена класса будет храниться состояние нашего счетчика. И опишем метод `next()`, который возвращает счетчик, увеличивая его значение на 1. Таким образом, счетчик инкапсулирован и никакой внешний код не может изменить его значение, а мы можем создать столько счетчиков, сколько нужно, в виде экземпляров этого класса (листинг 6.6).

Листинг 6.6. Объектно-ориентированный счетчик

```
class Counter {
    private n: number = 1;

    next(): number {
        return this.n++;
    }
}

let counter1: Counter = new Counter();
let counter2: Counter = new Counter();

console.log(counter1.next());
console.log(counter2.next());
console.log(counter1.next());
console.log(counter2.next());
```

Значение счетчика теперь является приватным членом класса

Можно создать несколько счетчиков

В результате выводится:
1
1
2
2

Такой подход более удачный. На самом деле большинство современных языков программирования предоставляют интерфейс для подобных нашему счетчику типов, выдающий значение при каждом вызове, со специальным синтаксисом для итерации. В TypeScript это можно сделать с помощью интерфейса `Iterable` и цикла `for ... of`. Мы рассмотрим данный вопрос далее, когда будем обсуждать обобщенное программирование. Пока отмечу, что это очень часто встречающийся паттерн. В C# он реализуется с помощью интерфейса `IEnumerable` и цикла `foreach`, а в Java — с помощью интерфейса `Iterable` и цикла `for : loop`.

Далее рассмотрим функциональный вариант реализации, в котором для реализации счетчика используются замыкания.

6.2.2. Функциональный счетчик

В листинге 6.7 мы реализуем функциональный счетчик с помощью функции `makeCounter()`, которая возвращает при вызове функцию-счетчик. Начальное значение счетчика будет задаваться в виде локальной (по отношению к функции `makeCounter()`) переменной, которую мы затем захватим в возвращаемой функции.

Листинг 6.7. Функциональный счетчик

```
type Counter = () => number;

function makeCounter(): Counter {
    let n: number = 1;

    return () => n++;
}

let counter1: Counter = makeCounter();
let counter2: Counter = makeCounter();

console.log(counter1());
console.log(counter2());
console.log(counter1());
console.log(counter2());
```

Тип Counter описан как функция, не принимающая аргументов и возвращающая number

Значение счетчика объявляется как переменная и захватывается лямбда-выражением

В результате выводится:
1
1
2
2

Теперь все счетчики представляют собой функции, так что вместо вызова `counter1.next()` мы вызываем просто `counter1()`. Как видите, каждый из счетчиков захватывает свое значение: вызов `counter1()` не влияет на вызов `counter2()`, поскольку при каждом вызове `makeCounter()` создается новая переменная `n`. У каждой возвращаемой функции — своя `n`. Счетчики являются замыканиями. Кроме того, эти значения сохраняются от вызова до вызова. В этом заключается отличие от поведения локальных для функции переменных, создаваемых при вызове функции и уничтожаемых при возврате из нее (рис. 6.5).

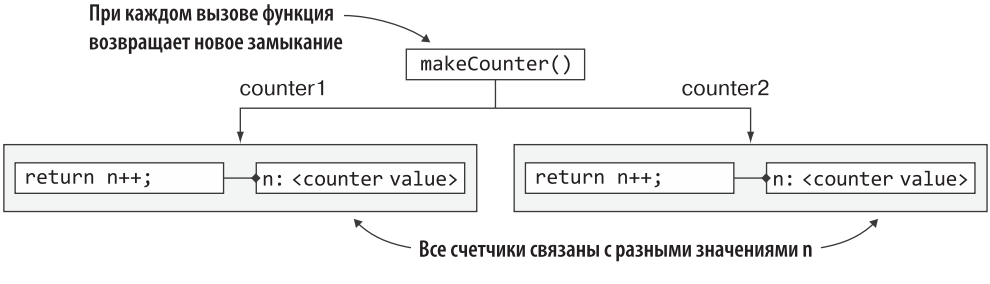


Рис. 6.5. Важно понимать, что у каждого замыкания (в нашем случае counter1 и counter2) — своя переменная n. При каждом вызове makeCounter() новой переменной n присваивается начальное значение 1, и она захватывается возвращаемым замыканием. А поскольку все значения разные, они не влияют друг на друга

6.2.3. Возобновляемый счетчик

Еще один способ описать счетчик — возобновляемая функция. Объектно-ориентированный счетчик отслеживает состояние в приватном члене класса. Функциональный отслеживает состояние в захваченном контексте.

ВОЗОБНОВЛЯЕМЫЕ ФУНКЦИИ

> Возобновляемой (resumable function) называется функция, которая отслеживает собственное состояние и при вызове не начинается с начального значения, а продолжает выполнение с того места, где произошел возврат из нее в прошлый раз.

В TypeScript вместо ключевого слова `return` для выхода из функции можно использовать ключевое слово `yield`, как показано в листинге 6.8. Это ключевое слово приводит к приостановке выполнения функции и возврату управления вызывающей стороне. При повторном вызове выполнение продолжается с кода, следующего за оператором `yield`.

Существует несколько ограничений по использованию `yield`: функцию следует объявить как генератор, а ее возвращаемым типом должна быть реализация интерфейса `IterableIterator`. Генераторы объявляются путем указания перед их названием символа *.

Листинг 6.8. Возобновляемый счетчик

```
function* counter(): IterableIterator<number> {
    let n: number = 1;

    while (true) {
        yield n++;
    }
}

let counter1: IterableIterator<number> = counter();
let counter2: IterableIterator<number> = counter();

console.log(counter1.next());
console.log(counter2.next());
console.log(counter1.next());
console.log(counter2.next());
```

Функция объявлена как генератор

Вместо return можно использовать yield

Наши счетчики — объекты, реализующие интерфейс IterableIterator

В результате выводится:
1
1
2
2

Эта реализация — нечто среднее между объектно-ориентированным и функциональным счетчиками. Реализация данного счетчика выглядит как функция: мы начинаем с `n=1` и выполняем бесконечный цикл, выдавая значение счетчика и увеличивая его на 1. С другой стороны, генерируемый компилятором код — объектно-ориентированный: фактически наш счетчик представляет собой `IterableIterator<number>`, и для получения следующего значения мы вызываем `next()`.

И хотя мы реализуем вышеописанное с помощью оператора `while (true)`, опасность застрять в бесконечном цикле нам не грозит; функция выдает значения и приостанавливается после каждого оператора `yield`. А компилятор скрыто транслирует написанный нами код в нечто напоминающее наши предыдущие реализации.

Тип данной функции — `() => IterableIterator<number>`. Обратите внимание: тот факт, что она является генератором, не влияет на ее тип. Тип функции без аргументов, возвращающей `IterableIterator<number>`, будет точно таким же. Компилятор на основе объявления `*` разрешает использовать операторы `yield`, но для системы типов это совершенно незаметно.

Мы еще вернемся к итераторам и генераторам в последующих главах и обсудим их подробнее.

6.2.4. Краткое резюме по реализациям счетчика

Прежде чем продолжить, коротко подытожим четыре способа реализации счетчика и различные языковые возможности, о которых было рассказано.

- Глобальный счетчик реализуется в виде простой функции, ссылающейся на глобальную переменную. У такого счетчика множество недостатков: значение счетчика не инкапсулировано должным образом и нельзя создать два отдельных экземпляра счетчика.
- Объектно-ориентированная реализация счетчика проста: значение счетчика — приватное состояние, для чтения и наращивания которого предоставляется метод `next()`. В большинстве языков программирования для подобных сценариев существуют интерфейсы наподобие `Iterable` и синтаксический сахар для работы с ними.
- Функциональный счетчик — это функция, возвращающая функцию. Возвращаемая функция и есть счетчик. В подобной реализации для хранения состояния счетчика используются возможности лямбда-захватов. Код более лаконичен, чем в объектно-ориентированной версии.
- В генераторах используется специальный синтаксис для создания возобновляемой функции. Вместо возврата из функции генератор производит выдачу значения; оно передается вызывающей стороне, но при этом отслеживается состояние на текущий момент и при последующих вызовах работа возобновляется с соответствующего места. Функция-генератор должна возвращать `IterableIterator`.

А теперь рассмотрим еще одну распространенную сферу применения функциональных типов данных: асинхронные функции.

6.2.5. Упражнения

1. Используя замыкания, реализуйте функцию, возвращающую при вызове следующее число в последовательности Фибоначчи.
2. Используя генератор, реализуйте функцию, возвращающую при вызове следующее число в последовательности Фибоначчи.

6.3. Асинхронное выполнение длительных операций

Приложения должны отличаться быстродействием и скоростью реакции, даже если часть операций совершается дольше. Последовательное выполнение всего кода привело бы к неприемлемым задержкам. Пользователи очень разочаруются, если приложение будет ждать завершения загрузки и не сможет из-за этого отреагировать на нажатие кнопки.

Как правило, не обязательно ждать завершения длительной операции, чтобы выполнить операцию, требующую меньше времени. Лучше выполнять подобные операции асинхронно; это позволит UI оставаться интерактивным во время загрузки. Асинхронность операций означает, что они не выполняются одна за другой, в том порядке, в котором встречаются в коде. Они могут работать параллельно, хотя и не обязательно так. JavaScript — однопоточный, поэтому среда выполнения использует цикл ожидания события для асинхронного выполнения операций. Мы обсудим в общих чертах как параллельное выполнение с помощью нескольких потоков, так и выполнение на основе цикла ожидания события при одном потоке. Но сначала рассмотрим пример, для которого может пригодиться асинхронное выполнение кода.

Допустим, нам нужно выполнить две операции: поприветствовать пользователей и перенаправить их на сайт www.weather.com, чтобы они могли посмотреть там текущую погоду. Для этого мы создадим две функции: `greet()` — запрашивает имя пользователя и приветствует его, и `weather()` — запускает браузер для просмотра текущей погоды. Сначала посмотрим на синхронную реализацию, а затем сравним ее с асинхронной.

6.3.1. Синхронная реализация

Мы реализуем функцию `greet()` с помощью пакета node `readline-sync`, как показано в листинге 6.9. Функция `question()` этого пакета обеспечивает возможность чтения входных данных из `stdin`. Она возвращает введенную пользователем символьную строку. Выполнение блокируется, пока пользователь не введет ответ и не нажмет Enter. Установить этот пакет можно с помощью команды `npm install –save readline-sync`.

Листинг 6.9. Синхронное выполнение

```
function greet(): void {
    const readlineSync = require('readline-sync');

    let name: string = readlineSync.question("What is your name? ");
    console.log(`Hi ${name}!`);
}

function weather(): void {
    const open = require('open');
    open('https://www.weather.com/');
}

greet();
weather();
```

Вызов question() блокирует выполнение до тех пор, пока пользователь не введет ответ

Сначала вызываем greet(), а потом weather()

Для реализации функции `weather()` мы воспользуемся пакетом `open` Node, с помощью которого можно открыть URL в браузере. Мы установим этот пакет с помощью команды `npm install --save open`.

Шаг за шагом обсудим, что происходит при работе этого кода. Сначала вызывается функция `greet()` и запрашивается имя пользователя. Выполнение приостанавливается до получения ответа от пользователя, после чего возобновляется и выводится приветствие. После возврата из функции `greet()` вызывается `weather()` и происходит переход на сайт www.weather.com.

Эта реализация работает, однако не является оптимальной. Две наши функции — приветствие пользователя и переход на сайт — в данном случае не зависят друг от друга, так что одна не может быть заблокирована, пока вторая не закончит работу. Можно вызвать эти функции в обратном порядке, поскольку запрос ввода пользователя явно требует больше времени, чем запуск приложения. Однако на практике не всегда можно с уверенностью сказать, какая из функций будет выполняться дольше. Лучше выполнять функции асинхронно.

6.3.2. Асинхронное выполнение: функции обратного вызова

Асинхронная версия функции `greet()` запрашивает имя пользователя, однако не блокирует выполнение в ожидании ответа. Оно продолжается, и вызывается `weather()`. Но мы все же хотели бы вывести имя пользователя после его получения, так что необходим способ уведомления о получении ответа от него. Для этого служат обратные вызовы.

Обратный вызов (callback) — это функция, передаваемая в асинхронную функцию в качестве аргумента. Асинхронная функция не блокирует выполнение; код продолжает работать строка за строкой. После завершения длительной операции (в данном случае ожидания ответа от пользователя с его именем) выполняется функция обратного вызова, и можно произвести нужные действия с результатом.

Посмотрим на асинхронную реализацию `greet()` в листинге 6.10. Мы воспользуемся предоставляемым Node модулем `readline`. В данном случае функция `question()` не блокирует выполнение, а принимает функцию обратного вызова в качестве аргумента.

Пройдемся по этой программе пошагово. Сразу после вызова функции `question()` и запроса имени у пользователя выполнение продолжается без какого-либо ожидания ответа пользователя; происходит возврат из `greet()` и вызов функции `weather()`. В результате запуска этой программы в терминале будет выведено `"What is your name?"`, но сайт www.weather.com будет открыт до ввода пользователем ответа.

При поступлении ответа от пользователя вызывается лямбда-выражение, которое выводит на экран приветствие с помощью вызова `console.log()` и закрывает интерактивный сеанс (так что пользователю больше не нужно вводить данные), задействовав `rl.close()`.

Листинг 6.10. Асинхронное выполнение с помощью обратного вызова

```
function greet(): void {
    const readline = require('readline');

    const rl = readline.createInterface({
        input: process.stdin,
        output: process.stdout
    });

    rl.question("What is your name? ", (name: string) => {
        console.log(`Hi ${name}!`);
        rl.close();
    });
}

function weather(): void {
    const open = require('open');
    open('https://www.weather.com/');
}

greet();
weather();
```

Используем модуль readline вместо модуля readline-sync

Метод createInterface() производит дополнительные настройки, необходимые для модуля readline. Для нашего примера они неважны

Функция обратного вызова представляет собой лямбда-выражение, получающее в качестве аргумента имя и выводящее его в консоль

6.3.3. Модели асинхронного выполнения

Как уже вкратце упоминалось в начале этого раздела, реализовать асинхронное выполнение можно с помощью потоков выполнения или цикла ожидания события в зависимости от того, как ваша среда выполнения и используемая вами библиотека реализуют асинхронные операции. В JavaScript асинхронные операции реализуются с помощью цикла ожидания событий.

Потоки выполнения

Любое приложение работает в виде процесса, работа которого начинается в основном потоке выполнения. Однако можно создать и несколько других потоков для выполнения кода. В таких POSIX-совместимых системах, как Linux и macOS, новые потоки выполнения создаются с помощью `pthread_create()`, а в Windows — `CreateThread()`. Эти API включены в саму операционную систему. А языки программирования предоставляют разработчику библиотеки с различными интерфейсами, которые, впрочем, все равно внутренним образом используют API операционной системы.

Различные потоки могут выполняться одновременно. Несколько ядер CPU могут выполнять инструкции параллельно, каждое — для своего потока. Если количество потоков превышает возможности аппаратного обеспечения, то операционная система обеспечивает равномерное разделение ресурсов CPU между потоками. Для этого

планировщик потоков приостанавливает и возобновляет потоки. Этот планировщик — ключевой компонент ядра операционной системы.

Мы не станем рассматривать примеры кода для потоков выполнения, поскольку JavaScript (а значит, и TypeScript), так уж исторически сложилось, ориентирован на однопоточное выполнение. В Node недавно появилась экспериментальная поддержка потоков-исполнителей, но эта возможность еще очень сырая на момент написания данной книги. Однако если вы пишете программы на других основных языках программирования, то, вероятно, умеете создавать новые потоки и выполнять в них код параллельно (рис. 6.6).

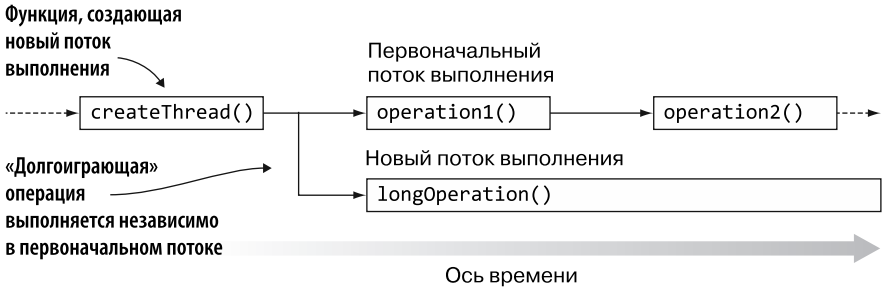


Рис. 6.6. Функция createThread() создает новый поток. Исходный продолжает выполнять operation1(), а затем operation2(), в то время как новый поток параллельно выполняет longRunningOperation()

Цикл ожидания событий

Вместо нескольких потоков выполнения можно использовать *цикл ожидания событий* (event loop). В нем используется очередь: асинхронные функции помещаются в нее, причем могут сами помещать туда другие функции. Первая функция в очереди удаляется из нее и выполняется, и так до тех пор, пока очередь не опустеет.

В качестве примера рассмотрим функцию обратного отсчета с заданного числа, показанную в листинге 6.11. Вместо блокировки выполнения до момента завершения обратного отсчета эта функция использует цикл ожидания событий и заносит в очередь еще один вызов себя же, пока не достигнет 0 (рис. 6.7).

Листинг 6.11. Обратный отсчет в цикле ожидания событий

```
type AsyncFunction = () => void;

let queue: AsyncFunction[] = [];

function countDown(counterId: string, from: number): void {
    console.log(`${counterId}: ${from}`);

    if (from > 0)
```

Ограничиваемся асинхронными функциями без аргументов, возвращающими void

Наша очередь представляет собой массив функций

Счетчик выводит идентификатор и текущее значение

```
        queue.push(() => countDown(counterId, from - 1));
    }

    queue.push(() => countDown('counter1', 4));

    while (queue.length > 0) {
        let func: AsyncFunction = <AsyncFunction>queue.shift();
        func();
    }
```

Если текущее значение больше 0, то счетчик заносит в очередь еще один вызов countDown(), уменьшая значение на 1

Запускаем процесс, занося в очередь вызов countDown() со значения 4

Пока в очереди содержатся функции, удаляем их оттуда по одной и выполняем

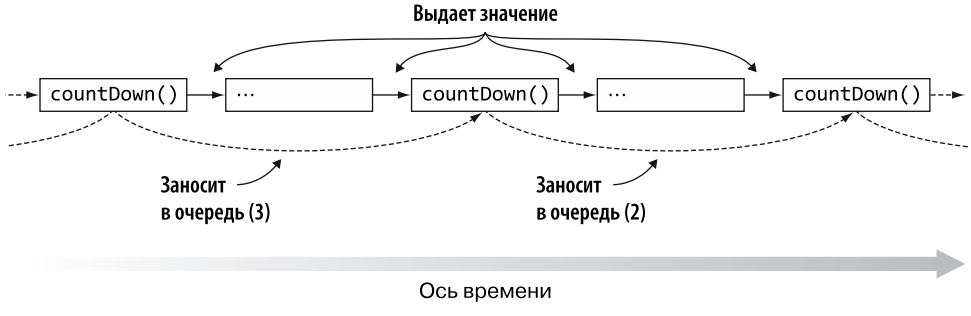


Рис. 6.7. Вызов countDown() отсчитывает один шаг, затем выдает значение и разрешает выполнение остального кода. А также заносит в очередь еще один вызов countDown() с уменьшенным значением счетчика. Если счетчик достиг 0, то countDown() не заносит в очередь еще один свой вызов

Результат выполнения этого кода будет выглядеть так:

```
counter1: 4
counter1: 3
counter1: 2
counter1: 1
counter1: 0
```

Достигнув 0, счетчик не заносит в очередь еще один вызов, так что выполнение программы прекращается. До сих пор это было не более интересно, чем, скажем, простой отсчет в цикле. Но что, если занести в очередь два счетчика (листинг 6.12)?

Листинг 6.12. Два счетчика в цикле ожидания событий

```
type AsyncFunction = () => void;

let queue: AsyncFunction[] = [];

function countDown(counterId: string, from: number): void {
    console.log(`${counterId}: ${from}`);

    if (from > 0)
```

```
        queue.push(() => countDown(counterId, from - 1));
}

queue.push(() => countDown('counter1', 4));
queue.push(() => countDown('counter2', 2));

while (queue.length > 0) {
    let func: AsyncFunction = <AsyncFunction>queue.shift();
    func();
}
```

Единственное отличие от предыдущего примера — мы занесли в очередь еще один счетчик

На этот раз результат выглядит так:

```
counter1: 4
counter2: 2
counter1: 3
counter2: 1
counter1: 2
counter2: 0
counter1: 1
counter1: 0
```

Как видим, на этот раз счетчики чередуются. Каждый из них отсчитывает один шаг, после чего получает возможность отсчитать второй. Добиться подобного при простом счете в цикле нам бы не удалось. Благодаря очереди каждая из двух функций выдает значение на каждом шаге обратного отсчета, позволяя другому коду выполняться перед своим следующим шагом отсчета.

Эти два счетчика не выполняются одновременно; то `counter1`, то `counter2` получает процессорное время. Но они выполняются асинхронно, то есть независимо друг от друга. Любой из них может завершить выполнение первым, вне зависимости от того, сколько еще после этого будет работать другой (рис. 6.8).

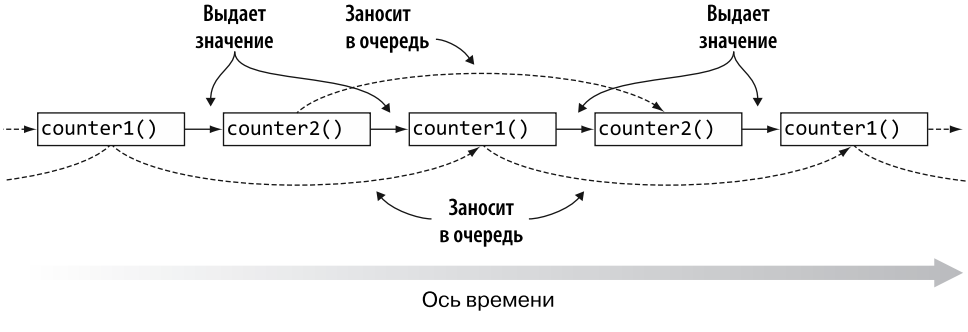


Рис. 6.8. Каждый из счетчиков запускается, после чего заносит в очередь следующую операцию. Выполнение происходит в порядке попадания операций в очередь. Все выполняется в одном потоке

Среда выполнения может обеспечить помещение в очередь операций, ожидающих ввода пользователем данных, например, с клавиатуры и отвечающих за

обработку этих данных только после их получения, что позволяет другому коду выполняться во время ожидания ввода. Благодаря этому можно разбить длительную операцию на две более короткие, первая из которых запрашивает входные данные и производит возврат, а вторая обрабатывает их после получения. За планирование выполнения второй операции после получения входных данных отвечает среда выполнения.

Циклы ожидания событий плохо подходят для длительных операций, которые нельзя разбить на несколько кусков. Если занести в очередь операцию, не возвращающую значение (`yield`) и выполняющуюся долгое время, то цикл ожидания событий может зависнуть вплоть до ее завершения.

6.3.4. Краткое резюме по асинхронным функциям

Если выполнять длительные операции синхронно, то никакой код не сможет выполняться, пока такая операция не закончится. Входные/выходные операции — отличные примеры длительных операций, поскольку приоритет чтения с диска или из сети выше, чем чтения из памяти.

Вместо синхронного выполнения подобных операций можно выполнять их асинхронно, передавая функцию обратного вызова, которую можно будет вызвать по завершении длительной операции. Существует две основные модели выполнения асинхронного кода: с помощью нескольких потоков выполнения и с использованием цикла ожидания событий.

Основное преимущество потоков — возможность параллельной работы на отдельных ядрах процессора, в результате чего различные части кода выполняются одновременно, а программа в целом завершается быстрее. Недостаток — передача данных между потоками требует затрат на тщательную синхронизацию. Мы не станем рассматривать этот вопрос в книге, но вы, вероятно, слышали о таких проблемах, как *взаимоблокировка* (deadlock) и *динамическая взаимоблокировка* (livelock), при которых выполнение двух потоков никогда не завершается, поскольку они ждут друг друга.

Цикл ожидания событий выполняется в одном потоке, но дает возможность поместить «долгоиграющий» код в конец очереди, пока тот ждет ввода данных. Преимущество цикла ожидания событий заключается в том, что не нужна синхронизация, поскольку все работает в одном потоке выполнения. Недостаток таков: несмотря на удобство помещения в очередь операций ввода/вывода, ожидающих данные, операции, требующие больших затрат ресурсов ЦП, все равно блокируют выполнение. Такую операцию, например сложные вычисления, нельзя занести в очередь, она ведь не ждет данных, а требует циклов ЦП. Потоки выполнения подходят для этой цели гораздо лучше.

Потоки применяются в большинстве основных языков программирования, JavaScript — заметное исключение. Несмотря на это, даже JavaScript сейчас находится в процессе добавления поддержки веб-потоков исполнителей (фоновых потоков

выполнения, работающих в браузере), а в Node появилась экспериментальная версия поддержки аналогичных потоков вне браузера.

Из следующего раздела вы узнаете, как можно сделать асинхронный код более понятным и читабельным.

6.3.5. Упражнения

1. Что из нижеприведенного позволит реализовать асинхронную модель выполнения?
   А. Потоки выполнения.
   Б. Цикл ожидания событий.
   В. Ни А, ни Б.
   Г. Как А, так и Б.
2. Могут ли две функции выполняться одновременно в асинхронной системе, в основе которой лежит цикл ожидания событий?
   А. Да.
   Б. Нет.
3. Могут ли две функции выполняться одновременно в асинхронной системе, в основе которой лежат потоки выполнения?
   А. Да.
   Б. Нет.

6.4. Упрощаем асинхронный код

Функции обратного вызова работают аналогично нашему счетчику из предыдущего примера. И если счетчик после каждого запуска заносил в очередь еще один вызов себя же, то асинхронная функция может принимать в качестве аргумента другую функцию и заносить в очередь ее вызов по завершении выполнения.

В качестве примера добавим в наш счетчик в листинге 6.13 обратный вызов, который будет заноситься в очередь при достижении счетчиком 0.

Функции обратного вызова — часто применяемый паттерн при работе с асинхронным кодом. В нашем примере используется функция обратного вызова без аргументов, но такие функции могут и получать аргументы от асинхронной функции. Так происходит в случае нашего вызова `question()` из модуля `readline`, в котором в функцию обратного вызова передавалась введенная пользователем строка.

Сцепление нескольких асинхронных функций с обратными вызовами приводит к множеству вложенных функций, как можно видеть в листинге 6.14, где мы запрашиваем имя пользователя, дату его рождения с помощью функций `getUserName()` и `getUserBirthday()` соответственно, его адрес и т. д. Функции зависят друг от друга, поскольку каждой из них требуется информация от предыдущей (например,

`getUserBirthday()` требуется имя пользователя). Все эти функции асинхронны, поскольку потенциально могут оказаться длительными, так что результаты возвращаются с помощью обратных вызовов. Мы воспользуемся этими обратными вызовами для вызова следующей функции в цепи.

Листинг 6.13. Счетчик с обратным вызовом

```
function countDown(counterId: string, from: number,
    callback: () => void): void {
    console.log(`${counterId}: ${from}`);

    if (from > 0)
        queue.push(() => countDown(counterId, from – 1, callback));
    else
        queue.push(callback);
}

queue.push(() => countDown('counter1', 4,
    () => console.log('Done')));
```

Добавляем в функцию аргумент обратного вызова — функцию без аргументов, возвращающую void

По завершении обратного отсчета заносим в очередь на выполнение обратный вызов

Передаем функцию обратного вызова, выводящую Done (Выполнено) по завершении выполнения счетчика

Листинг 6.14. Организация цепи обратных вызовов

```
declare function getUserName(
    callback: (name: string) => void): void;
declare function getUserBirthday(name: string,
    callback: (birthday: Date) => void): void;
declare function getUserEmail(birthday: Date,
    callback: (email: string) => void): void;

getUserName((name: string) => {
    console.log(`Hi ${name}!`);
    getUserBirthday(name, (birthday: Date) => {
        const today: Date = new Date();
        if (birthday.getMonth() == today.getMonth() &&
            birthday.getDay() == today.getDay())
            console.log('Happy birthday!');

        getUserEmail(birthday, (email: string) => {
            /* ... */
        });
    })
});
```

Мы не станем приводить здесь реализации этих функций, покажем только объявления

Функция обратного вызова для getUserName() вызывает getUserBirthday()

Функция обратного вызова для getUserBirthday() вызывает getUserEmail() и т. д.

В обратном вызове, который вызывается при получении функцией `getUserName()` имени пользователя, мы запускаем функцию `getUserBirthday()`, передавая в нее это имя. В обратном вызове, который вызывается при получении функцией `getUserBirthday()` даты рождения пользователя, мы запускаем `getUserEmail()`, передавая в нее дату рождения и т. д.

Мы не станем обсуждать сами реализации всех функций `getUser...` из этого примера, поскольку они аналогичны реализации функции `greet()` из предыдущего раздела. Нас здесь больше интересует общая структура вызовов кода.

Организованный подобным образом код сложно читать, ведь чем больше функций обратного вызова мы добавляем в цепь, тем больше получаем вложенных лямбда-выражений внутри лямбда-выражений. Оказывается, что для этого паттерна вызовов асинхронных функций существует лучшая абстракция: промисы.

6.4.1. Сцепление промисов

Мы начнем с такого факта: функция, подобная `getUserName(callback: (name: string) => void)`, представляет собой асинхронную функцию, которая в какой-то момент времени определит имя пользователя и передаст его в заданную нами функцию обратного вызова. Другими словами, `getUserName` «обещает»[1] в конце концов вернуть строку с именем. Обратите также внимание: мы хотим, чтобы при получении «обещанного» значения функция вызывала другую функцию, передавая это значение в качестве аргумента.

ПРОМИСЫ И ФУНКЦИИ-ПРОДОЛЖЕНИЯ

> Промис (promise) — объект-заместитель для значения, которое окажется доступным в некий момент в будущем. Еще до выполнения кода, выдающего это значение, другой код сможет использовать промис, чтобы подготовить обработку значения после его поступления, задать действия в случае ошибки и даже отменить это будущее выполнение. Функция, которая должна выполняться при появлении результата промиса, называется функцией-продолжением или просто продолжением (continuation).

Две основные составные части промиса — это значение некоего типа `T`, который наша функция «обещает» предоставить нам, и возможность задать функции из `T` в некий другой тип `U` (`(value: T) => U`), которая будет вызвана при осуществлении промиса и получении значения. Это альтернатива передачи обратного вызова непосредственно в функцию.

Для начала модифицируем объявления функций в листинге 6.15, чтобы вместо получения аргумента — обратного вызова они возвращали объект `Promise`. Функция `getUserName()` будет возвращать `Promise<string>`, `getUserBirthday()` — возвращать `Promise<Date>`, а `getUserEmail()` — тоже `Promise<string>`.

Листинг 6.15. Функции, возвращающие промисы

```
declare function getUserName(): Promise<string>;
declare function getUserBirthday(name: string): Promise<Date>;
declare function getUserEmail(birthday: Date): Promise<string>;
```

В JavaScript (а значит, и в TypeScript) есть реализующий эту абстракцию встроенный тип `Promise<T>`. В C# ее реализует `Task<T>`, а в Java аналогичную функциональность предоставляет класс `CompletableFuture<T>`.

---

[1] Автор обыгрывает английское слово to promise — «обещать». В русскоязычной литературе устоялся термин «промис» для обозначения понятия promise. — *Примеч. пер.*

У промиса есть метод `then()`, в который можно передать функцию-продолжение. Каждая функция `then()` возвращает другой промис, так что вызовы `then()` можно сцепить. Это позволяет избавиться от вложенности, свойственной реализации на основе обратных вызовов (листинг 6.16).

Листинг 6.16. Организация цепи промисов

```
getUserName()
    .then((name: string) => {
        console.log(`Hi ${name}!`);

     return getUserBirthday(name);
    })
    .then((birthday: Date) => {
        const today: Date = new Date();
        if (birthday.getMonth() == today.getMonth() &&
            birthday.getDay() == today.getDay())
            console.log('Happy birthday!');
        return getUserEmail(birthday);
    })
    .then((email: string) => {
        /* ... */
    });
```

Вызываем метод then() промиса, возвращаемого функцией getUserName()

Используем в этой функции-продолжении значение, возвращаемое функцией getUserBirthday()

А поскольку метод then() тоже возвращает промис, можно вызвать метод then() возвращаемого значения...

...и еще раз

Как вы можете видеть, вместо многократно вложенных друг в друга обратных вызовов функции-продолжения сцеплены более читабельным образом: запускается функция, после чего[1] запускается следующая и т. д.

6.4.2. Создание промисов

Использовать этот паттерн следует, разобравшись в создании промисов. Общая идея проста, хотя и основана на функциях высшего порядка: промис принимает в качестве аргумента функцию, которая принимает в качестве аргумента другую функцию, — на первый взгляд, разобраться в этом непросто.

Промис, служащий для получения значения определенного типа, например `Promise<string>`, на самом деле не знает, как вычислить это значение. Он предоставляет метод `then()`, который позволяет сцепить продолжения, как мы видели выше, но не может вычислить эту строку. В случае `getUserName()` обещанная строка представляет собой имя пользователя, а в случае `getUserEmail()` — адрес электронной почты. Как же при таких условиях обобщенный `Promise<string>` смог бы определить это значение? А он и не может сам по себе. Конструктор промиса принимает в качестве аргумента функцию, которая на самом деле и вычисляет значение. В случае `getUserName()` она запрашивает у пользователя его имя и получает ответ. А затем промис уже может применить эту функцию: вызвать ее напрямую, занести в очередь

[1] Автор обыгрывает значение английского слова then, означающего «затем, после, далее». — *Примеч. пер.*

для цикла ожидания событий или запланировать ее выполнение в потоке в зависимости от реализации, различающейся в разных языках и библиотеках.

Пока все в порядке. Промис `Promise<string>` получает код, который выдает значение. Но, поскольку данный код может быть запущен в любое время, необходим механизм, с помощью которого код мог бы сообщить промису о наличии значения. Для этой цели промис передает в указанный код функцию `resolve()`. После выяснения значения код вызывает `resolve()` и передает значение в промис (рис. 6.9).

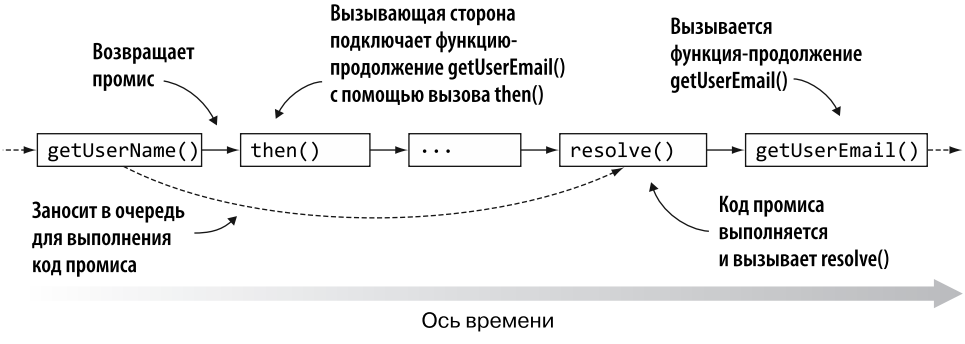


Рис. 6.9. Функция getUserName() заносит в очередь код получения имени пользователя и возвращает Promise<string>. Вызывающий getUserName() код может затем вызвать для промиса метод then(), чтобы подключить в качестве продолжения функцию getUserEmail() — код, который можно выполнять только при наличии имени пользователя. Позднее, в какой-то момент времени код получения имени запускается и вызывает функцию resolve(), передавая в него это имя. На данном этапе вызывается функция-продолжение getUserEmail(), теперь уже с доступным именем пользователя

Посмотрим теперь в листинге 6.17, как реализовать функцию `getUserName()` так, чтобы возвращать промис.

Листинг 6.17. Возвращающая промис функция getUserName()

```
function getUserName(): Promise<string> {
    return new Promise<string>(
        (resolve: (value: string) => void) => {
        const readline = require('readline');

        const rl = readline.createInterface({
            input: process.stdin,
            output: process.stdout
        });

        rl.question("What is your name? ", (name: string) => {
            rl.close();
            resolve(name);
        });
    });
}
```

Передаем лямбда-выражение в конструктор Promise, ожидающий в качестве аргумента функцию resolve()

Читаем строку из stdin с помощью того же кода, что и в функции greet()

Наконец, получив имя, вызываем переданную функцию resolve(), передавая в нее это имя

Метод `getUserName()` просто создает и возвращает промис. Тот инициализируется функцией, принимающей аргумент `resolve` типа `(value: string) => void`. Эта функция включает код запроса у пользователя его имени и, получив его, вызывает `resolve()` для передачи значения промису.

Если реализовать длительные операции так, чтобы они возвращали промисы, то можно сцепить асинхронные вызовы с помощью `Promise.then()`, что значительно повысит читабельность кода.

6.4.3. И еще о промисах

Функции-продолжения — далеко не единственная возможность промисов. Рассмотрим обработку ошибок с помощью промисов и еще пару способов выстраивания последовательности их выполнения, помимо использования `then()`.

Обработка ошибок

Промис может находиться в одном из трех состояний: ожидающий выполнения, завершенный и отклоненный. *Ожидание выполнения* (pending) означает, что промис был создан, но пока еще не разрешен (то есть переданная функция, которая отвечает за получение значения, еще не вызвала `resolve()`). *Завершенным* (settled) является такой промис, когда `resolve()` уже была вызвана и значение получено; этап, на котором вызываются функции-продолжения. Но что будет в случае ошибки? Если отвечающая за предоставление значения функция генерирует исключение, то промис переходит в состояние *отклоненного* (rejected).

На самом деле отвечающая за предоставление значения функция может принимать дополнительную функцию-аргумент, которая позволяет перевести промис в отклоненное состояние и указать причину этого. Вместо того чтобы передать конструктору:

```
(resolve: (value: T) => void) => void
```

вызывающая сторона может передать:

```
(resolve: (value: T) => void, reject: (reason: any) => void) => void
```

Второй аргумент представляет собой функцию типа `(reason: any) => void`, позволяющую указать для промиса `reason` любого типа и пометить его как отклоненный.

Промис автоматически считается отклоненным при генерации функцией исключения даже без вызова `reject()`. Помимо функции `then()`, у любого промиса доступна функция `catch()`, которой можно передать функцию-продолжение, для вызова при отклонении промиса по какой-либо причине (рис. 6.10).

Расширим в листинге 6.18 нашу функцию `getUserName()` так, чтобы она отклоняла пустую строку.

Отклоняется (путем вызова `reject` или из-за генерации ошибки) не только текущий промис, но и все промисы в цепи, привязанные к нему с помощью `then()`. Если любой из промисов в цепи вызовов `then()` отклоняется, то вызывается функция-продолжение `catch()`, добавленная в конец цепи.

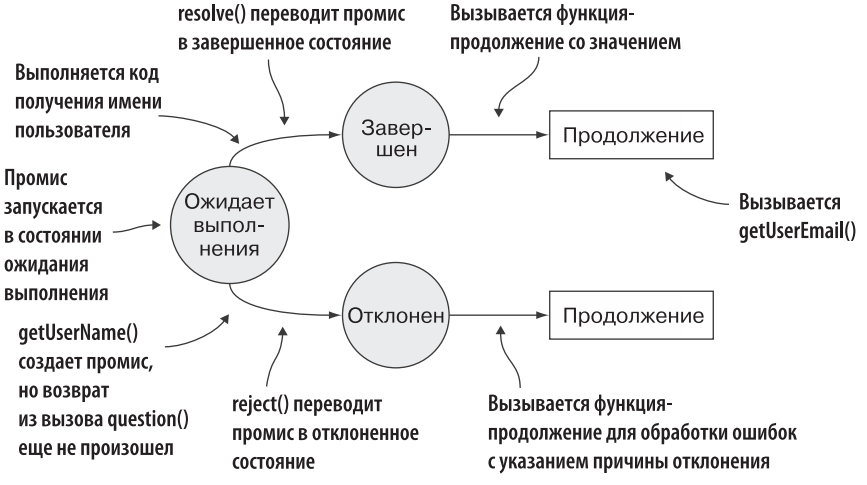


Рис. 6.10. Промис запускается в состоянии ожидания выполнения. (Функция getUserName() запланировала выполнение кода для опроса пользователя, но функция question() еще ничего не вернула.) Функция resolve() переводит его в завершенное состояние и вызывает функцию-продолжение, если та была задана (после ввода пользователем имени). Значение уже доступно, так что можно вызвать функцию-продолжение (в данном случае getUserEmail()). Функция reject() переводит промис в отклоненное состояние и вызывает функцию-продолжение для обработки ошибок, если та была задана. Значение недоступно, но вместо него передается причина ошибки

Листинг 6.18. Отклонение промиса

```
function getUserName(): Promise<string> {
    const readline = require('readline');

    const rl = readline.createInterface({
        input: process.stdin,
        output: process.stdout
    });

    return new Promise<string>(
        (resolve: (value: string) => void,
         reject: (reason: string) => void) => {
        rl.question("What is your name? ", (name: string) => {
            rl.close();

            if (name.length != 0) {
                resolve(name);
            } else {
                reject("Name can't be empty");
            }
        });
    });
}

getUserName()
    .then((name: string) => {console.log(`Hi ${name}!`); })
    .catch((reason: string) => {console.log(`Error: ${reason}`); });
```

Указываем дополнительный аргумент reject

Отклоняем промис, если значение name.length равно 0

Подключаем новую функцию-продолжение; при отклонении промиса (или генерации ошибки) вызывается catch()

Организация цепи синхронных функций

Функции-продолжения можно сцепить не только описанными выше способами. Начнем с того, что функция-продолжение не обязана возвращать промис. Кроме того, не всегда в цепь объединяются асинхронные функции: встречаются и быстро выполняемые функции-продолжения, которые можно выполнять синхронным образом. Еще раз взглянем на наш исходный пример в листинге 6.19, в котором все функции-продолжения возвращали промисы.

Листинг 6.19. Цепь возвращающих промисы функций

```
getUserName()
    .then((name: string) => {
        console.log(`Hi ${name}!`);
        return getUserBirthday(name);
    })
    .then((birthday: Date) => {
        const today: Date = new Date();
        if (birthday.getMonth() == today.getMonth() &&
            birthday.getDay() == today.getDay())
            console.log('Happy birthday!');
        return getUserEmail(birthday);
    })
    .then((email: string) => {
        /* ... */
    });
```

Функция getUserName() возвращает Promise<string>

Функция getUserBirthday() возвращает Promise<Date>

Функция getUserEmail() возвращает Promise<string>

В данном случае все функции должны выполняться асинхронно, поскольку ждут ввода пользователем данных. Но что, если, получив имя пользователя, мы просто хотим вставить его в строку и вернуть результат? Если наша функция-продолжение — просто `` `Hi ${name}!` ``, то она возвращает строку, а не промис. Ничего страшного, функция `.then()` автоматически преобразует ее в `Promise<string>` для дальнейшей обработки следующей функцией-продолжением, как показано в листинге 6.20.

Листинг 6.20. Сцепление функций, не возвращающих промисы

```
getUserName()
    .then((name: string) => {
        return `Hi ${name}!`;
    })
    .then((greeting: string) => {
        console.log(greeting);
    });
```

В данном случае промис не возвращается, но функция then() преобразует результат в Promise<string>

Интуитивно это представляется логичным: даже если функция-продолжение возвращает обычную строку, она все равно включена в цепь за промисом, вследствие чего не будет выполнена сразу же. Таким образом, она фактически является промисом, завершаемым после завершения исходного промиса.

Другие способы сочетания промисов

До сих пор мы рассматривали метод `then()` (и `catch()`), связывающий промисы так, что они завершаются по очереди, друг за другом. Существует еще два способа планирования выполнения асинхронных функций: с помощью `Promise.all()` и `Promise.race()` — статических методов класса `Promise`. Метод `Promise.all()` принимает в качестве аргументов набор промисов и возвращает промис, завершаемый при завершении *всех* указанных промисов. Метод `Promise.race()` принимает набор промисов и возвращает промис, завершаемый при завершении *любого из* указанных промисов.

Метод позволяет `Promise.all()` планировать выполнение набора независимых асинхронных функций, например извлечение сообщений входящей почты пользователей из базы данных и изображений профиля из CDN с последующей передачей обоих значений в UI, как показано в листинге 6.21. Не имеет смысла планировать последовательное выполнение этих функций изображения одна за другой, поскольку они не зависят друг от друга. С другой стороны, нам нужно собрать их результаты и передать другой функции.

Листинг 6.21. Установка последовательности выполнения с помощью метода Promise.all()

```
class InboxMessage {/* ... */ }
class ProfilePicture {/* ... */ }

declare function getInboxMessages(): Promise<InboxMessage[]>;
declare function getProfilePicture(): Promise<ProfilePicture>;
declare function renderUI(
    messages: InboxMessage[], picture: ProfilePicture): void;

Promise.all([getInboxMessages(), getProfilePicture()])
  .then((values: [InboxMessage[], ProfilePicture]) => {
      renderUI(values[0], values[1]);
  });
```

Функции getInboxMessages() и getProfilePicture() — независимые асинхронные

Для renderUI() нужен результат обеих функций

Метод Promise.all() создает промис, завершаемый после разрешения промисов обеих указанных функций

values представляет собой кортеж, содержащий оба результата

Передаем извлеченные значения в функцию renderUI()

Реализовать подобный паттерн с помощью обратных вызовов значительно сложнее, поскольку не существует механизма их соединения.

Рассмотрим пример применения метода `Promise.race()` в листинге 6.22. Допустим, профиль пользователя реплицирован на два узла. Попробуем извлечь его из обоих и задействуем тот результат, который будет возвращен быстрее. В этом случае можно продолжать выполнять программу сразу после получения результата с любого из узлов.

Реализовать такой сценарий с помощью обратных вызовов без промисов еще сложнее (рис. 6.11).

Промисы — это понятная абстракция для выполнения асинхронных функций. Благодаря планированию обработки с помощью методов `then()` и `catch()` не только

код становится более читабельным, чем при использовании обратных вызовов, но и появляется возможность обработки ошибок, а также соединения нескольких промисов с помощью методов `Promise.all()` и `Promise.race()`. Библиотеки для работы с промисами доступны в большинстве основных языков программирования, и предоставляемая ими функциональность примерно одинакова, хотя названия методов могут слегка различаться (например, `race()` в некоторых языках называется `any()`).

Листинг 6.22. Установка последовательности выполнения с помощью метода Promise.race()

```
class UserProfile {/* ... */ }

declare function getProfile(node: string): Promise<UserProfile>;

declare function renderUI(profile: UserProfile): void;

Promise.race([getProfile("node1"), getProfile("node2")])
    .then((profile: UserProfile) => {
        renderUI(profile);
    });
```

Вызываем функцию getProfile() по одному разу для каждого узла

В функцию-продолжение передается наш единственный UserProfile — тот, который был быстрее получен

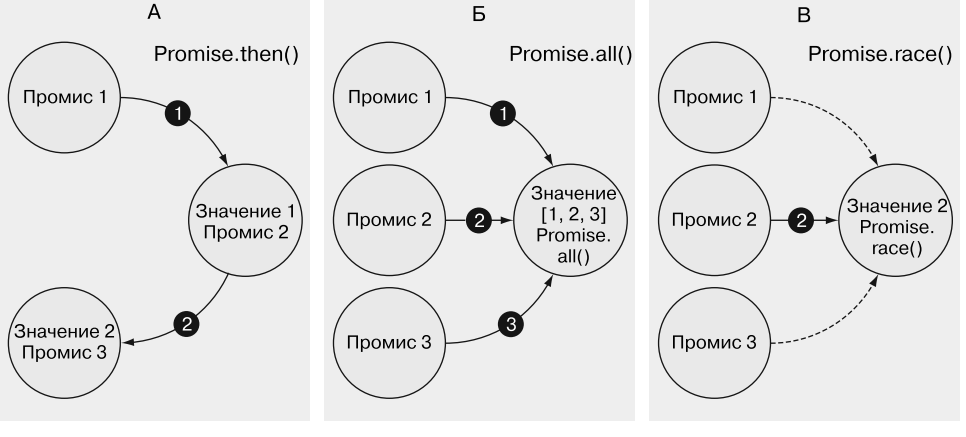


Рис. 6.11. Различные способы сочетания промисов. А: промис 1 завершается и передает значение 1 промису 2; промис 2 завершается и передает значение 2 промису 3. Б: промис 1, промис 2 и промис 3 завершаются. Когда они все переходят в состояние завершенных, Promise.all() получает все их значения и может продолжать работу, завершая ее собственным значением. В: один из промисов завершается первым (в данном случае промис 2). Метод Promise.race() получает значение 2 и может продолжать работу, завершая ее собственным значением

Вот практически и все, чем библиотеки могут помочь нам в написании ясного асинхронного кода. Повышение читабельности такого кода требует изменений синтаксиса самого языка. Аналогично тому, как оператор `yield` позволяет проще описывать функции-генераторы, синтаксис многих языков расширен ключевыми словами `async` и `await`, упрощающими написание асинхронных функций.

6.4.4. async/await

Промисы позволяли нам запрашивать у пользователей различную информацию, и мы упорядочивали наши запросы с помощью функций-продолжений. Взглянем еще раз на эту реализацию в листинге 6.23. Мы обернем ее в функцию `getUserData()`.

Листинг 6.23. Краткий обзор сцепления промисов

```
function getUserData(): void {
    getUserName()
        .then((name: string) => {
            console.log(`Hi ${name}!`);
            return getUserBirthday(name);
        })
        .then((birthday: Date) => {
            const today: Date = new Date();
            if (birthday.getMonth() == today.getMonth() &&
                birthday.getDay() == today.getDay())
                console.log('Happy birthday!');
            return getUserEmail(birthday);
        })
        .then((email: string) => {
            /* ... */
        });
}
```

Обратите внимание снова: все функции-продолжения принимают в качестве аргумента значение того же типа, что и у промиса из предыдущей функции. Конструкция `async/await` позволяет лучше выразить это в коде. Можно провести параллель с генераторами и синтаксисом `*/yield`, который обсуждался в предыдущем разделе.

Элемент `async` — ключевое слово, указываемое перед ключевым словом `function` аналогично тому, как в генераторах перед `function` указывается `*`. И подобно тому, как `*` можно использовать, только если функция возвращает `Iterator`, слово `async` можно указывать лишь для функций, которые возвращают `Promise`. Как и `*`, `async` не влияет на тип функции. Тип функций `function getUserData(): Promise<string>` и `async function getUserData(): Promise<string>` одинаков: `() => Promise<string>`. И аналогично тому, как `*` указывает, что функция является генератором, позволяя вызывать внутри нее `yield`, так и `async` дает знать, что функция асинхронна, позволяя вызывать внутри нее `await`.

Слово `await` можно указывать перед возвращающей промис функцией, чтобы получить значение, возвращаемое по завершении этого промиса. Вместо того чтобы писать `getUserName().then ((name: string) => { /* … */ })`, можно написать `let name: string = await getUserName()`. Прежде чем разбираться, как этот код работает, посмотрим, как написать функцию `getUserData()` с помощью `async` и `await`.

Сразу видно, что написанная подобным образом функция `getUserData()` намного более удобна для чтения, чем вариант со сцеплением промисов с помощью метода `then()`. Компилятор генерирует точно такой же код; внутренний механизм

не отличается. Это просто более удобный способ выразить ту же самую цепь функций-продолжений. Он позволяет написать весь код в одной функции, указывая ключевое слово `await`. При этом программа будет ждать результатов всех прочих возвращающих промисы функций, вместо того чтобы помещать все продолжения в отдельные функции, соединяемые с помощью `then()`.

Каждое слово `await` эквивалентно помещению следующего за ним кода в продолжение метода `then()`: это позволяет писать меньше лямбда-выражений, а асинхронный код читается подобно синхронному. Что касается `catch()`, если возвращать нечего, например, в случае исключения, то в вызове `await` генерируется исключение, которое затем можно перехватить с помощью обычного оператора `try/catch`. Достаточно просто обернуть вызов `await` в блок `try` для перехвата возможных ошибок (листинг 6.24).

Листинг 6.24. Использование конструкции async/await

```
async function getUserData(): Promise<void> {
    let name: string = await getUserName();
    console.log(`Hi ${name}!`);

    let birthday: Date = await getUserBirthday(name);
    const today: Date = new Date();
    if (birthday.getMonth() == today.getMonth() &&
        birthday.getDay() == today.getDay())
        console.log('Happy birthday!');

    let email: string = await getUserEmail(birthday);
    /* ... */
}
```

Функция getUserData() должна возвращать промис, поскольку отмечена как асинхронная

Можно использовать эту строку с именем в той же функции

Ждем, пока getUserName() завершится и выдаст строку с именем

Ждем, пока getUserBirthday() завершится и выдаст строку с датой рождения

Делаем то же самое для getUserEmail(): ждем завершения промиса и получения строкового значения

6.4.5. Краткое резюме по понятному асинхронному коду

Вкратце подытожим описанные в этом разделе подходы к написанию асинхронного кода. Мы начали с обратных вызовов — передачи в асинхронную функцию функции обратного вызова, вызываемой по завершении операции. Этот подход вполне работоспособный, но приводит к множеству вложенных обратных вызовов, сильно затрудняющих анализ кода. Кроме того, при этом очень сложно соединить несколько независимых асинхронных функций, если для продолжения работы необходимы результаты от них всех.

Далее мы поговорили о промисах. Это абстракция для написания асинхронного кода. Они помогают планировать выполнение кода (в языках, где применяются потоки, выполнение планируется по ним) и позволяют указывать функции-продолжения, вызываемые по завершении (получении значения) или отклонении (возникновении ошибки) промиса. Методы `Promise.all()` и `Promise.race()` позволяют по-разному соединять наборы промисов.

Наконец, синтаксис `async/await`, обычный ныне для большинства основных языков программирования, позволяет писать асинхронный код в даже еще более понятном виде, читаемый совершенно как обычный код. Вместо указания функции-продолжения с помощью метода `then()` указывается ключевое слово `await`. Программа будет ждать результата промиса и продолжения выполнения с этого места. Выполняемый компьютером внутренний код — тот же самый, но читать такой синтаксис намного приятнее.

6.4.6. Упражнения

1. С какого состояния начинает работу промис?
   А. Завершенный.
   Б. Отклоненный.
   В. Ожидающий выполнения.
   Г. С любого из них.
2. Какой из следующих методов позволяет добавить в цепь функцию-продолжение, вызываемую в случае отклонения промиса?
   А. `then()`.
   Б. `catch()`.
   В. `all()`.
   Г. `race()`.
3. Какой из следующих методов позволяет добавить в цепь функцию-продолжение, вызываемую в случае завершения всего набора промисов?
   А. `then()`.
   Б. `catch()`.
   В. `all()`.
   Г. `race()`.

Резюме

- Замыкание — это лямбда-выражение, сохраняющее, помимо прочего, фрагмент состояния содержащей его функции.
- Простой паттерн проектирования «Декоратор» можно реализовать с помощью замыкания и захвата декорированной функции вместо реализации отдельной новой функции.
- Счетчик можно реализовать с помощью замыкания, которое отслеживает состояние счетчика.
- Написанный с применением синтаксиса `*/yield` генератор является возобновляемой функцией.
- Длительные операции желательно делать асинхронными, чтобы они не блокировали выполнение остальной программы.

- Две основные модели асинхронного выполнения: потоки и циклы ожидания событий.
- Обратный вызов — функция, передаваемая в асинхронную функцию и вызываемая по ее завершении.
- Промисы — распространенная абстракция для выполнения асинхронных функций, имеет функции-продолжения как альтернативу обратным вызовам. Состояния промиса таковы: ожидающий выполнения, завершенный (значение получено) и отклоненный (возникла ошибка).
- Статические методы `Promise.all()` и `Promise.race()` — механизмы соединения набора промисов различным образом.
- Конструкция `async/await` — современный синтаксис написания кода на основе промисов подобно синхронному коду.

Теперь, обсудив детально все приложения функциональных типов данных, от основ передачи функций в качестве аргументов и до генераторов и асинхронных функций, мы можем перейти к следующей большой теме: подтипам. Как мы увидим в главе 7, подтипы — это отнюдь не только наследование.

Ответы к упражнениям

6.1. Простой паттерн проектирования «Декоратор»

Одна из возможных реализаций, возвращающая функцию, которая добавляет в обернутую фабрику журналирование:

```
function loggingDecorator(factory: () => Widget): () => Widget {
    return () => {
        console.log(«Widget created»);
        return factory();
    }
}
```

6.2. Реализация счетчика

1. Вариант реализации с помощью замыкания, захватывающего переменные `a` и `b` из функции-обертки:

```
function fib(): () => number {
    let a: number = 0;
    let b: number = 1;

    return () => {
        let next: number = a;
        a = b;
        b = b + next;
        return next;
    }
}
```

2. Вариант реализации с помощью генератора, выдающего следующее число в последовательности:

```
function *fib2(): IterableIterator<number> {
    let a: number = 0;
    let b: number = 1;

    while (true) {
        let next: number = a;
        a = b;
        b = a + next;
        yield next;
    }
}
```

6.3. Асинхронное выполнение длительных операций

1. Г — как потоки, так и цикл ожидания событий можно использовать для асинхронного выполнения кода.
2. Б — цикл ожидания событий не выполняет код параллельно. Функции попадают в очередь и выполняются асинхронно, но не одновременно.
3. А — параллельное выполнение возможно с помощью потоков; несколько потоков могут выполнять несколько функций одновременно.

6.4. Упрощаем асинхронный код

1. В — промис начинает работу с состояния ожидания выполнения.
2. Б — для добавления в цепь функции-продолжения, вызываемой в случае отклонения промиса, используется метод `catch()`.
3. В — для добавления в цепь функции-продолжения, вызываемой в случае завершения всех промисов, используется метод `all()`.

7 Подтипизация

В этой главе

- Устранение неоднозначностей типов в TypeScript.
- Безопасная десериализация.
- Значения на случай ошибки.
- Совместимость типов для типов-сумм, коллекций и функций.

Мы уже рассмотрели простые типы, их сочетания, а также функциональные типы данных. Настало время взглянуть еще на один аспект систем типов: отношения между типами. В этой главе мы познакомим вас с отношением «тип — подтип». Хотя вы, возможно, знакомы с этой концепцией по объектно-ориентированному программированию, мы не станем говорить в данной главе о наследовании, а сосредоточим внимание на иных приложениях подтипизации.

Сначала мы поговорим о том, что такое подтипизация, и о двух способах ее реализации в языках программирования: структурном и номинальном. Затем снова обратимся к нашему примеру с Mars Climate Orbiter и объясним трюк с `unique symbol`, которым мы воспользовались в главе 4 при обсуждении типобезопасности.

А поскольку тип может быть подтипом другого типа и у него самого могут быть подтипы, мы взглянем на все это с точки зрения иерархии типов, в которой обычно есть один тип в самом верху и иногда один тип в самом низу. Мы рассмотрим возможности использования этого высшего типа в сценариях наподобие десериализации, когда обычно доступно не слишком много информации о типе. Мы также научимся задействовать низший тип в качестве значения на случай ошибки.

Во второй половине этой главы мы научимся устанавливать более сложные отношения «тип — подтип». Это поможет нам понять, какие значения можно заменять какими. Нужно ли реализовывать адаптеры или можно просто передать значение другого типа как есть? Каково будет отношение «тип — подтип» между коллекциями двух типов, один из которых является подтипом другого? А функциями, принимающими или возвращающими аргументы этих типов? Мы посмотрим на простом примере, как передавать геометрические фигуры в виде типов-сумм, коллекций и функций, — процесс, известный под названием *«вариантность»* (variance). Мы изучим различные разновидности вариантности. Но сначала разберемся, что такое подтипы в TypeScript.

7.1. Различаем схожие типы в TypeScript

Большинство примеров данной книги, хоть и написаны на TypeScript, не содержат ничего специфического для этого языка, их можно переписать на большинстве других основных языков программирования. Данный раздел — исключение: мы обсудим специфическую для TypeScript методику, поскольку она позволит плавно перейти к обсуждению подтипов.

Вернемся к примеру с фунт-силами/ньютонами на секунду из главы 4. Напомню, что мы смоделировали в нем две различные единицы измерения в виде двух отдельных классов. Нам требовалась гарантия того, что модуль проверки типов не разрешит принять значение одного типа за значение другого, поэтому мы воспользовались `unique symbol` с целью устранить двусмысленность. Мы не вдавались ранее в подробности того, для чего это было нужно, так что сделаем это теперь, в листинге 7.1.

Листинг 7.1. Типы для фунт-силы в секунду и ньютона на секунду

```
declare const NsType: unique symbol;

class Ns {
    value: number;
    [NsType]: void;

    constructor(value: number) {
        this.value = value;
    }
}

declare const LbfsType: unique symbol;

class Lbfs {
    value: number;
    [LbfsType]: void;

    constructor(value: number) {
        this.value = value;
    }
}
```

Объявляем NsType как уникальный символ и добавляем свойство [NsType] типа void в класс Ns

Объявляем LbfsType как уникальный символ и добавляем свойство [LbfsType] типа void в класс Lbfs

Если опустить эти два объявления, то произойдет интересная вещь: можно будет передавать объект `Ns` вместо `Lbfs` и наоборот, причем компилятор никаких ошибок не выдаст. Реализуем для демонстрации этого процесса функцию `acceptNs()`, ожидающую аргумент типа `Ns`. А затем попробуем передать в `acceptNs()` объект типа `Lbfs` (листинг 7.2).

Листинг 7.2. Фунт-силы на секунду и ньютоны на секунду без уникальных символов

```
class Ns {
    value: number;

    constructor(value: number) {
        this.value = value;
    }
}

class Lbfs {
    value: number;

    constructor(value: number) {
        this.value = value;
    }
}

function acceptNs(momentum: Ns): void {
    console.log(`Momentum: ${momentum.value}Ns`);
}

acceptNs(new Lbfs(10));
```

Типы Ns и Lbfs больше не включают свойство unique symbol

Функция acceptNs() принимает в качестве аргумента объект Ns и выводит в журнал его значение

Передаем экземпляр Lbfs в функцию acceptNs()

Как ни странно, этот код работает и выводит в журнал `Momentum: 10 Ns` — явно не то, что нам нужно. Мы объявили два отдельных типа именно во избежание путаницы двух единиц измерения и аварии Mars Climate Orbiter. Что же произошло? Нам придется разобраться в нюансах подтипизации.

ПОДТИПИЗАЦИЯ

> Тип S — подтип типа T, если экземпляр S можно безопасно использовать везде, где ожидается экземпляр T.

Это нестрогая формулировка знаменитого *принципа подстановки*[1] *Барбары Лисков* (Liskov substitution principle). Два типа связаны отношением «подтип — надтип», если можно использовать экземпляр подтипа везде, где ожидается экземпляр надтипа, не меняя код.

Существует два способа задать отношения «тип — подтип». Первый из них, используемый в большинстве основных языков программирования (например, Java и C#), называется *номинальной подтипизацией* (nominal subtyping). Тип является подтипом другого типа, если описан как таковой явным образом, с помощью

[1] Иногда его также называют принципом замены Лисков. — *Примеч. пер.*

синтаксиса вида `class Triangle extends Shape`. После этого можно применять экземпляр `Triangle` везде, где ожидается экземпляр `Shape` (например, в качестве аргумента функции). Если же не объявить, что тип `Triangle` расширяет `Shape`, то компилятор не позволит использовать его вместо `Shape`.

Напротив, *структурная подтипизация* (structural subtyping) не требует явного указания отношения «подтип» в коде. Экземпляр типа, например `Lbfs`, можно использовать вместо другого типа, например `Ns`, если он включает все члены класса, объявленные в этом другом типе. Иными словами, если структура типа аналогична другому типу (те же члены класса и, возможно, еще какие-либо дополнительные члены), то он автоматически считается подтипом этого другого типа.

НОМИНАЛЬНАЯ И СТРУКТУРНАЯ ПОДТИПИЗАЦИЯ

При номинальной подтипизации тип считается подтипом другого типа, если это отношение описано в коде явным образом. При структурной подтипизации тип считается подтипом другого типа, если включает все члены надтипа и, возможно, некие дополнительные.

В отличие от C# и Java, в TypeScript используется структурная подтипизация. Именно поэтому, если в классах `Ns` и `Lbfs` объявлен только член `value` типа `number`, их можно использовать друг вместо друга.

7.1.1. Достоинства и недостатки номинальной и структурной подтипизации

Довольно часто структурная подтипизация удобна, так как позволяет задавать отношения между типами вне нашего контроля. Допустим, мы используем библиотеку, в которой объявлен тип `User` с полями `name` и `age`. В нашем коде объявлен интерфейс `Named`, который требует от реализующих его типов наличия свойства `name`. Мы можем использовать экземпляр `User` везде, где ожидается `Named`, хоть `User` и не реализует `Named` явным образом, как показано в листинге 7.3 (в объявлении класса `User` не указано `class User implements Named`).

Если бы требовалось явно объявить, что `User` реализует `Named`, то мы столкнулись бы с проблемой, ведь тип `User` взят из внешней библиотеки. Мы не можем менять ее код, поэтому пришлось бы идти окольным путем и объявить новый тип, расширяющий `User` и реализующий `Named` (`class NamedUser extends User implements Named {}`), просто чтобы связать эти два типа. Если же система типов использует структурную подтипизацию, то можно этого не делать.

С другой стороны, в некоторых случаях крайне нежелательно, чтобы тип считался подтипом другого типа на основании одной их структуры. Например, нельзя допускать использования экземпляров `Lbfs` вместо `Ns`. Именно так по умолчанию ведет себя система типов при номинальной подтипизации, что позволяет легко избежать ошибок. В то же время структурная подтипизация требует от нас более тщательной проверки того, что значение — именно того типа, который мы ждем,

а не типа со схожей структурой. В подобных случаях структурная подтипизация намного предпочтительнее.

Листинг 7.3. Класс User — структурный подтип интерфейса Named

```
/* код из библиотеки */
class User {
    name: string;
    age: number;

    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
    }
}

/* наш код */
interface Named {
    name: string;
}

function greet(named: Named): void {
    console.log(`Hi ${named.name}!`);
}

greet(new User("Alice", 25));
```

User — тип из внешней библиотеки, недоступный для модификации

Функция greet() ожидает в качестве аргумента экземпляр типа, соответствующего интерфейсу Named

Экземпляр класса User можно передать вместо Named

Существует несколько методов, позволяющих задействовать номинальную подтипизацию в TypeScript, например используемый по всей этой книге трюк с `unique symbol`. Изучим его более внимательно.

7.1.2. Моделирование номинальной подтипизации в TypeScript

В нашем случае с `Ns`/`Lbfs` мы фактически пытаемся смоделировать номинальную подтипизацию. Нам нужно, чтобы компилятор считал тип подтипом `Ns`, только когда мы объявим его так явным образом, а не из-за наличия у него члена класса `value`.

Для этого необходимо добавить в `Ns` член класса, который не сможет случайно объявить ни один другой класс. В TypeScript выражение `unique symbol` позволяет сгенерировать «название», гарантированно уникальное в пределах всего кода. Различные объявления `unique symbol` сгенерируют различные «названия», и никакое объявленное пользователем название не совпадет со сгенерированным.

Мы объявили уникальный символ для типа `Ns` как `NsType`. Объявление уникального символа выглядит следующим образом: `declare const NsType: unique symbol` (как в листинге 7.1). Объявив уникальное название, можно создать свойство с этим названием, указав его в квадратных скобках. Необходимо описать тип для данного свойства, но мы не собираемся присваивать ему никаких значений, поскольку свойство служит лишь для различения типов. А поскольку фактическое его значение нас

не волнует, лучше всего для этой цели подходит единичный тип данных. Так что мы воспользовались типом `void`.

То же самое мы проделали с типом `Lbfs`, и теперь у этих двух типов различные структуры: в одном из них есть свойство `[NsType]`, а в другом — `[LbfsType]`, как показано в листинге 7.4. А благодаря `unique symbol` невозможно случайно описать свойство с тем же названием в другом типе. Теперь единственный способ создать подтип `Ns` или `Lbfs` — выполнить наследование от них явным образом.

Листинг 7.4. Моделирование номинальной подтипизации

```
declare const NsType: unique symbol;

class Ns {
    value: number;
    [NsType]: void;

    constructor(value: number) {
        this.value = value;
    }
}

declare const LbfsType: unique symbol;

class Lbfs {
    value: number;
    [LbfsType]: void;

    constructor(value: number) {
        this.value = value;
    }
}

function acceptNs(momentum: Ns): void {
    console.log(`Momentum: ${momentum.value}Ns`);
}

acceptNs(new Lbfs(10));
```

Этот код больше не компилируется

Если попытаться передать экземпляр `Lbfs` вместо `Ns`, будет выдана следующая ошибка: `Argument of type 'Lbfs' is not assignable to parameter of type 'Ns'. Property '[NsType]' is missing in type 'Lbfs' but required in type 'Ns'` (Невозможно присвоить аргумент типа `'Lbfs'` параметру типа `'Ns'`. В типе `'Lbfs'` отсутствует требуемое для `Ns` свойство `'[NsType]'`).

В этом разделе было дано определение подтипизации и рассказано о двух способах установить отношение «тип — подтип» между двумя типами: номинально (путем явного объявления) или структурно (на основе одинаковой структуры типов). Мы также видели, что, невзирая на использование в TypeScript структурной подтипизации, можно легко смоделировать там номинальную с помощью уникальных символов в тех случаях, когда структурная неуместна.

7.1.3. Упражнения

1. Является ли в TypeScript тип `Painting` подтипом `Wine`, если они описаны следующим образом?

```
class Wine {
    name: string;
    year: number;
}

class Painting {
    name: string;
    year: number;
    painter: Painter;
}
```

2. Является ли в TypeScript тип `Car` подтипом `Wine`, если они описаны следующим образом?

```
class Wine {
    name: string;
    year: number;
}

class Car {
    make: string;
    model: string;
    year: number;
}
```

7.2. Присваиваем что угодно, присваиваем чему угодно

Теперь, когда мы уже знаем, что такое подтипизация, рассмотрим два предельных случая: тип, присваивающий что угодно, и тип, присваиваемый чему угодно. В первом из них можно хранить абсолютно любое значение. Второй пригоден к использованию вместо любого другого типа, если его экземпляр недоступен.

7.2.1. Безопасная десериализация

Мы обсудили типы `any` и `unknown` в главе 4. В типе `unknown` может храниться значение любого другого типа. Мы уже упоминали, что в других объектно-ориентированных языках программирования обычно есть тип `Object`, ведущий себя аналогично. На самом деле и в TypeScript есть тип `Object`, включающий несколько часто используемых методов наподобие `toString()`. Но это далеко не все, как мы увидим в данном разделе.

Тип `any` — более опасный. Можно не только ему присваивать любое значение, но и присваивать значение типа `any` любому другому типу данных, обходя тем самым проверку типов. Он обеспечивает взаимодействие с кодом на JavaScript, но последствия его использования могут оказаться весьма неожиданными. Допустим, у нас есть функция, которая десериализует объект с помощью стандартного метода `JSON.parse()`, как показано в листинге 7.5. А поскольку `JSON.parse()` — это функция языка JavaScript, с которым взаимодействует TypeScript, она не является строго типизированной, тип возвращаемого ею значения — `any`. Представим, что нам нужно десериализовать экземпляр класса `User`, у которого есть свойство `name`.

Листинг 7.5. Десериализуем any

```
class User {
    name: string;                          ← У класса User есть свойство name

    constructor(name: string) {
        this.name = name;
    }
}

function deserialize(input: string): any {   ← Функция deserialize() — простой
    return JSON.parse(input);                  адаптер для JSON.parse(),
}                                              возвращающий значение типа any

function greet(user: User): void {           ← В функции greet() используется
    console.log(`Hi ${user.name}!`);           свойство name заданного объекта User
}

greet(deserialize('{"name": "Alice" }'));    ← Десериализуем JSON-строку
greet(deserialize('{}'));                      для корректного объекта User
                                             ← Но можно десериализовать и объект, не являющийся User
```

Последний вызов функции `greet()` выведет в журнал `"Hi undefined!"`, поскольку тип `any` обходит проверку типов и компилятор позволяет нам работать с возвращаемым значением как значением типа `User`, хотя оно таковым и не является. Результат явно неидеальный. Необходимо проверять правильность типа данных перед использованием функции `greet()`.

В данном случае необходимо проверить, есть ли у нашего объекта свойство `name` типа `string` (листинг 7.6). В нашем случае этого достаточно для приведения данного объекта к типу `User`. Желательно также проверить, что наш объект не `null` и не `undefined` — два специальных типа в TypeScript. Для этого можно, например, добавить в наш код подобную проверку, вызывая ее перед вызовом `greet()`. Обратите внимание: данная проверка типов производится во время выполнения, поскольку зависит от входного значения, и статически обеспечить ее соблюдение невозможно.

Возвращаемый тип функции `isUser` — `user is User` — специфика синтаксиса TypeScript, но, я надеюсь, более или менее понятная. Этот тип во многом напоминает возвращаемый тип `boolean`, однако несет для компилятора дополнительный смысл. Если функция возвращает `true`, то тип переменной `user` — `User`, и компи-

лятор может использовать эту информацию на вызывающей стороне. Фактически в каждом блоке `if`, в котором функция `isUser` вернула `true`, у переменной `user` тип `User` вместо `any`.

Листинг 7.6. Проверка типов во время выполнения для объекта User

```
class User {
    name: string;

    constructor(name: string) {
        this.name = name;
    }
}

function deserialize(input: string): any {
    return JSON.parse(input);
}

function greet(user: User): void {
    console.log(`Hi ${user.name}!`);
}

function isUser(user: any): user is User {
    if (user === null || user === undefined)
        return false;

    return typeof user.name === 'string';
}

let user: any = deserialize('{"name": "Alice" }');
if (isUser(user))
    greet(user);

user = undefined;
if (isUser(user))
    greet(user);
```

Эта функция проверяет, относится ли заданный аргумент к типу User. Мы считаем, что к нему относится переменная со свойством name типа string

Перед каждым использованием объекта user проверяем, есть ли у него свойство name типа string

Вполне работоспособный подход. При запуске этого кода выполняется только первый вызов с именем пользователя `Alice`. Второй не будет выполнен, поскольку в данном случае у `user` нет свойства `name`. Впрочем, у этого подхода есть недостаток: ничто не обязывает нас реализовать данную проверку. А потому мы вполне можем случайно забыть вызвать функцию проверки, в результате чего произвольный результат из функции `deserialize()` попадет в функцию `greet()` и ничто ему в этом не помешает.

Хорошо было бы найти возможность сказать компилятору: «Этот объект может относиться абсолютно к любому типу», но без дополнительного «Верьте мне, я знаю, что делаю», которое подразумевает `any`. Нам нужен другой тип — надтип для любого типа в системе, чтобы любое возвращаемое `JSON.parse()` значение было его подтипом. Таким образом, система типов будет проверять, добавили ли мы нужную проверку типов перед приведением к типу `User`.

ВЫСШИЙ ТИП

Тип, которому можно присвоить любое значение, называется также высшим (top type), поскольку все остальные типы являются его подтипами. Другими словами, этот тип располагается вверху иерархии подтипов (рис. 7.1).

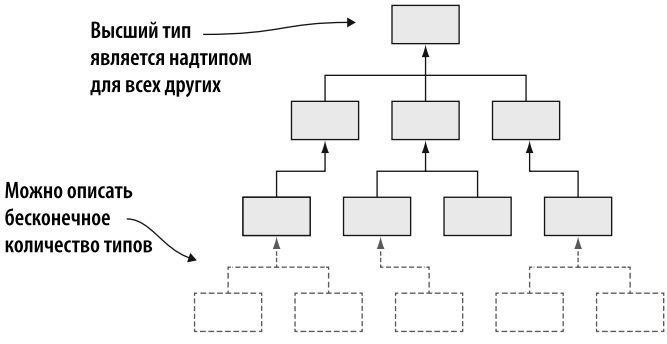


Рис. 7.1. Высший тип является надтипом для любого другого типа. Можно описать произвольное количество типов, но любой из них будет подтипом высшего типа. Там, где ожидается значение высшего типа, можно использовать значение любого типа

Модифицируем нашу реализацию. Начнем с типа `Object` — надтипа *большинства* типов системы типов, за исключением `null` и `undefined`. В системе типов TypeScript есть несколько замечательных средств безопасности, одно из которых таково: значения `null` и `undefined` находятся вне областей определения прочих типов. Помните врезку «Ошибка стоимостью миллиард долларов» из главы 3 (подраздел 3.2.2 «Опциональные типы данных», пункт «Опционал своими руками»)? В большинстве языков программирования можно присвоить `null` переменной любого типа. В TypeScript это не разрешается, если использовать флаг компилятора `--strictNullChecks` (что я вам настоятельно рекомендую). TypeScript считает, что `null` — значение типа `null`, а `undefined` — значение типа `undefined`. Как следствие, наш высший тип, надтип всего на свете, является суммой этих типов: `Object | null | undefined`. На самом деле уже есть готовый тип, описанный подобным образом: `unknown`. Перепишем наш код, воспользовавшись `unknown`, как показано в листинге 7.7, после чего обсудим различия между `any` и `unknown`.

Изменения малозаметны, но весьма серьезны: получив значение из функции `JSON.parse()`, мы сразу же преобразуем его из `any` в `unknown`. Это безопасно, поскольку преобразовать в `unknown` можно что угодно. Типом аргумента функции `isUser` остается `any` ради упрощения реализации. (Выполнить проверку вида `typeof user.name` для типа `unknown`, не прибегая к дополнительному приведению типов, нельзя.)

Код работает, как и раньше, различие лишь в том, что код перестанет компилироваться, если удалить любой из вызовов функции `isUser()`. Компилятор выдаст при этом следующую ошибку: `Argument of type 'unknown' is not assignable to parameter of type 'User'` (Невозможно присвоить аргумент типа `'unknown'` параметру типа `'User'`).

Листинг 7.7. Более строгая типизация с помощью типа unknown

```
class User {
    name: string;

    constructor(name: string) {
        this.name = name;
    }
}

function deserialize(input: string): unknown {
    return JSON.parse(input);
}

function greet(user: User): void {
    console.log(`Hi ${user.name}!`);
}

function isUser(user: any): user is User {
    if (user === null || user === undefined)
        return false;

    return typeof user.name === 'string';
}

let user: unknown = deserialize('{"name": "Alice" }');
if (isUser(user))
    greet(user);

user = deserialize("null");
if (isUser(user))
    greet(user);
```

Функция deserialize() теперь возвращает тип unknown

Типом аргумента функции isUser остается any

Объявляем нашу переменную с типом unknown

Нельзя просто передать переменную типа `unknown` функции `greet()`, ожидающей в качестве аргумента `User`. На помощь приходит функция `isUser()`, поскольку компилятор автоматически считает нашу переменную относящейся к типу `User`, когда она возвращает `true`.

При такой реализации забыть проверить тип просто невозможно; компилятор этого не позволит. Он разрешит использовать наш объект как объект типа `User` только после того, как мы подтвердим, что `user is User`.

РАЗЛИЧИЯ МЕЖДУ UNKNOWN И ANY

Можно присваивать что угодно как переменным unknown, так и переменным any, однако существуют различия в использовании переменных этих типов. Значение типа unknown можно применять в качестве значения какого-либо типа (например, User), только убедившись, что значение действительно относится к этому типу (подобно тому как мы сделали с функцией, возвращавшей user как User). Значение же типа any можно сразу же использовать как значение любого другого. Тип any обходит проверку типов.

В других языках программирования используются иные механизмы для определения того, относится ли значение к заданному типу. Например, в C# есть ключевое слово `is`, а в Java — `instanceof`. В общем случае при работе со значением, которое может относиться к *любому* типу, мы начинаем с того, что считаем его значением высшего типа. А затем проводим соответствующие проверки с целью убедиться, что оно относится к нужному нам типу, прежде чем произвести понижающее приведение к этому типу.

7.2.2. Значения на случай ошибки

Теперь посмотрим на противоположную задачу: тип, который можно использовать вместо любого другого. Возьмем простой пример из листинга 7.8. В этой игре космический корабль поворачивается влево (`Left`) или вправо (`Right`). Эти возможные направления мы представим в виде перечисляемого типа данных. Наша задача — реализовать функцию, принимающую на входе направление и преобразующую его в угол поворота космического корабля. А поскольку необходимо учесть все случаи, мы генерируем ошибку, если в функцию передано значение, отличное от двух ожидаемых значений `Left` и `Right`.

Листинг 7.8. Использование функции turnAngle для преобразования направления в угловое значение

```
enum TurnDirection {
    Left,
    Right
}

function turnAngle(turn: TurnDirection): number {
    switch (turn) {
        case TurnDirection.Left: return -90;
        case TurnDirection.Right: return 90;
        default: throw new Error("Unknown TurnDirection");
    }
}
```

Преобразуем поворот налево (Left) в угол –90 градусов; а поворот направо (Right) — в угол 90 градусов

Генерируем ошибку, если встретилось непредвиденное значение

Пока все нормально. Но что, если создать функцию для обработки ошибок? Например, функцию журналирования ошибки перед ее генерацией. Эта функция всегда будет генерировать ошибку, поэтому можно объявить ее как возвращающую тип `never`, как мы видели в главе 2. Напомню, что `never` — пустой тип данных, которому нельзя присвоить никакое значение. С его помощью мы продемонстрируем: наша функция никогда не возвращает значения либо потому, что работает бесконечно, либо потому, что генерирует ошибку, как показано в листинге 7.9.

Листинг 7.9. Выдача отчета об ошибке

```
function fail(message: string): never {
    console.error(message);
    throw new Error(message);
}
```

Выводим ошибку в консоль, после чего генерируем ошибку

Функция fail() никогда ничего не возвращает (всегда генерирует ошибку), так мы объявляем ее с возвращаемым типом never

Если заменить оператор `throw` в функции `turnAngle` на функцию `fail()`, то получим примерно следующее (листинг 7.10).

Листинг 7.10. Функция turnAngle(), в которой используется fail()

```
function turnAngle(turn: TurnDirection): number {
    switch (turn) {
        case TurnDirection.Left: return -90;
        case TurnDirection.Right: return 90;
        default: fail("Unknown TurnDirection");
    }
}
```

Заменяем оператор throw вызовом fail()

Этот код почти работает, но не совсем. При компиляции с использованием флага `--strict` выдается следующая ошибка: `Function lacks ending return statement and return type does not include "undefined"` (У функции отсутствует завершающий оператор `return`, а возвращаемый тип не содержит `"undefined"`).

Компилятор не находит оператор `return` в ветке `default` и считает это ошибкой. Для решения данной проблемы можно, например, возвращать фиктивное значение, как показано в листинге 7.11, ведь мы все равно сгенерируем ошибку, прежде чем дойдем до этого оператора `return`.

Листинг 7.11. Функция turnAngle(), использующая fail() и возвращающая фиктивное значение

```
enum TurnDirection {
    Left,
    Right
}

function turnAngle(turn: TurnDirection): number {
    switch (turn) {
        case TurnDirection.Left: return -90;
        case TurnDirection.Right: return 90;
        default: {
            fail("Unknown TurnDirection");
            return -1;
        }
    }
}
```

Никогда не возвращаемое (поскольку fail() сгенерирует ошибку) фиктивное значение

Но если в какой-то момент в будущем мы поменяем функцию `fail()` таким образом, что она не всегда будет генерировать ошибку? Тогда наш код вернет фиктивное значение, хотя не должен этого делать. Существует решение получше: вернуть результат вызова `fail()`, как показано в листинге 7.12.

Листинг 7.12. Функция turnAngle(), использующая fail() и возвращающая результат ее выполнения

```
function turnAngle(turn: TurnDirection): number {
    switch (turn) {
        case TurnDirection.Left: return -90;
        case TurnDirection.Right: return 90;
        default: return fail("Unknown TurnDirection");
    }
}
```

Просто возвращает то, что вернула fail()

Этот код работает потому, что `never` — не просто тип, у которого нет значений, а еще и подтип всех остальных типов в системе.

НИЗШИЙ ТИП ДАННЫХ

Тип, который является подтипом для любого из прочих типов данных системы, называется низшим (bottom type), поскольку располагается в самом низу иерархии подтипов. Чтобы быть подтипом всех остальных типов, он должен включать все члены всех прочих типов. А поскольку количество типов и их членов бесконечно, низший тип должен включать бесконечное количество членов, что невозможно. Поэтому низший тип всегда пустой: тип, создать реальное значение которого невозможно (рис. 7.2).

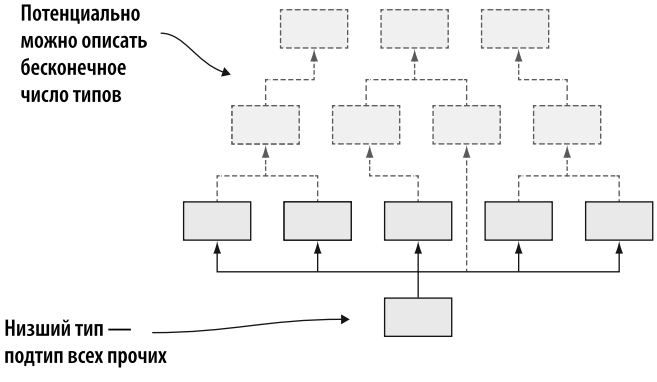


Рис. 7.2. Низший тип — подтип всех прочих типов системы. Можно описать сколько угодно различных типов, но все они будут надтипами для низшего. Можно передавать его значение всюду, где требуется значение любого типа (хотя создать такое значение невозможно)

А поскольку всегда можно присвоить значение `never` как низшего типа любому другому, то можно и вернуть его из функции. Компилятор не станет против этого возражать, ведь речь идет о повышающем приведении типов (преобразовании значения подтипа в надтип), которое допустимо производить неявным образом. Мы говорим компилятору: «Преобразуй вот это значение, которое нельзя создать, в строку». Это допустимо. Поскольку функция `fail()` никогда не возвращает значений, нам никогда не придется на самом деле преобразовывать что-либо в строку.

Такой подход лучше, чем предыдущий, ведь если мы модифицируем функцию `fail()` так, что в некоторых случаях она перестанет генерировать ошибку, то компилятор заставит нас исправить код. Для начала он потребует, чтобы мы поменяли возвращаемый тип `fail()` с `never` на что-то другое, например `void`. А затем заметит, что мы пытаемся передать это значение в виде объекта `string`, не проходящего проверку типов. Нам придется поменять реализацию `turnAngle()`, например вернуть обратно явный вызов оператора `throw`.

Низший тип позволяет притвориться, будто у нас есть значение какого-либо типа, даже если на самом деле у нас его нет.

7.2.3. Краткое резюме по высшим и низшим типам

Коротко резюмируем рассмотренные в этом разделе вопросы. Два типа могут находиться в отношении «тип — подтип», при котором один из них является надтипом, а второй — подтипом. Предельные случаи этого — надтип всех прочих типов и подтип всех прочих типов.

В надтипе всех прочих типов — высшем — можно хранить значение любого другого. В TypeScript этот тип называется `unknown`. Он удобен, в частности, при работе с данными, которые могут оказаться чем угодно, например прочитанным из NoSQL базы данных JSON-документом. Изначально для таких данных указывается высший тип, а затем производятся проверки, необходимые для приведения к типу, с которым уже можно работать.

Подтип всех прочих типов — низший — используется для возврата значения любого другого типа. В TypeScript этот тип называется `never`. Один из примеров его применения — возвращаемый тип значения для функции, которая ничего не может вернуть, поскольку всегда генерирует исключение.

Обратите внимание: хотя высший тип данных есть в большинстве основных языков программирования, лишь в немногих из них есть низший. В нашей самодельной реализации из главы 2 мы создали пустой, однако не низший тип. Невозможно создать пользовательский низший тип, если это не предусмотрено компилятором.

7.2.4. Упражнения

1. Можно ли инициализировать значение `x` типа `number` результатом функции `makeNothing()`, возвращающей `never` (без приведения типов)?

```
declare function makeNothing(): never;

let x: number = makeNothing();
```

2. Можно ли инициализировать значение `x` типа `number` результатом функции `makeSomething()`, возвращающей `unknown` (без приведения типов)?

```
declare function makeSomething(): unknown;

let x: number = makeSomething();
```

7.3. Допустимые подстановки

На данный момент мы рассмотрели несколько простых примеров подтипизации. Мы отметили, в частности, что если класс `Triangle` расширяет класс `Shape`, то `Triangle` — подтип `Shape`. Теперь попробуем ответить на несколько более хитрых вопросов.

- Каково отношение «тип — подтип» между типами-суммами `Triangle | Square` и `Triangle | Square | Circle`?
- Каково отношение «тип — подтип» между массивом треугольников (`Triangle[]`) и массивом геометрических фигур (`Shape[]`)?

- Каково отношение «тип — подтип» для обобщенной структуры данных, например `List<T>`, между `List<Triangle>` и `List<Shape>`?
- А между функциональными типами данных `() => Shape` и `() => Triangle`?
- И наоборот, как насчет функционального типа `(argument: Shape) => void` и функционального типа `(argument: Triangle) => void`?

Получить ответы на эти вопросы важно для того, чтобы выяснить, вместо каких из этих типов можно подставить их подтипы. Мы должны понимать, можем ли передать заданной функции, ожидающей аргумент одного из этих типов, один из его подтипов взамен.

Сложность в предыдущих примерах состоит в том, что это не просто `Triangle extends Shape`. Речь идет о типах, заданных *на основе* `Triangle` и `Shape`: частей типов-сумм, типов элементов коллекций, типов аргументов функций или возвращаемых ими типов.

7.3.1. Подтипизация и типы-суммы

Сначала рассмотрим самый простой пример: тип-сумму. Возьмем функцию `draw()`, которая может отрисовывать `Triangle`, `Square` или `Circle`. Можем ли мы передать в нее `Triangle` или `Square`? Как вы, вероятно, догадываетесь, ответ — да, можем. Как видно из листинга 7.13, такой код скомпилируется.

Листинг 7.13. Triangle | Square вместо Triangle | Square | Circle

```
declare const TriangleType: unique symbol;
class Triangle {
    [TriangleType]: void;
    /* члены класса Triangle */
}

declare const SquareType: unique symbol;
class Square {
    [SquareType]: void;
    /* члены класса Square */
}

declare const CircleType: unique symbol;
class Circle {
    [CircleType]: void;
    /* члены класса Circle */
}

declare function makeShape(): Triangle | Square;
declare function draw(shape: Triangle | Square | Circle): void;

draw(makeShape());
```

Функция makeShape() возвращает Triangle или Square (реализацию мы опустим)

Функция draw() принимает в качестве аргумента Triangle, Square или Circle (реализацию мы опустим)

В этих примерах используется номинальная подтипизация, поскольку мы не приводим для этих типов полную реализацию. На практике их отличали бы разнообразные свойства и методы. Мы моделируем эти различные свойства в наших примерах с помощью уникальных символов. Ведь если оставить содержимое классов пустым, то они все оказались бы эквивалентными вследствие структурной подтипизации языка TypeScript.

Как мы и ожидали, данный код прекрасно скомпилируется. А наоборот — нет. Если мы можем нарисовать `Triangle` или `Square` и попытаемся нарисовать `Triangle`, `Square` или `Circle`, то компилятор нам этого не позволит, поскольку мы можем случайно передать объект `Circle` в функцию `draw()`, которая не будет знать, что с ним делать. Можете сами убедиться, что следующий код (листинг 7.14) не компилируется.

Листинг 7.14. Triangle | Square | Circle как Triangle | Square

```
declare function makeShape(): Triangle | Square | Circle;
declare function draw(shape: Triangle | Square): void;

draw(makeShape());
```

Этот код больше не компилируется

Мы поменяли местами типы, так что makeShape() может также возвращать Circle, в то время как draw() больше не принимает Circle

`Triangle | Square` — подтип `Triangle | Square | Circle`: всегда можно подставить `Triangle` или `Square` вместо `Triangle`, `Square` или `Circle`, но не наоборот.

Интуитивно это непонятно, поскольку `Triangle | Square` — нечто «меньшее», чем `Triangle | Square | Circle`. При использовании наследования у подтипа оказывается больше свойств, чем у надтипа. Для типов-сумм справедливо обратное: надтип содержит больше типов, чем подтип (рис. 7.3).

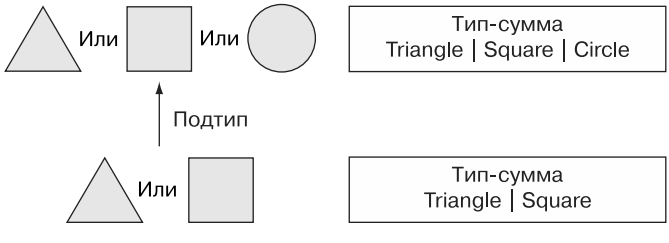


Рис. 7.3. Triangle | Square — подтип Triangle | Square | Circle, поскольку везде, где ожидается тип Triangle, Square или Circle, можно использовать тип Triangle или Square

Пусть тип `EquilateralTriangle` наследует `Triangle`, как показано в листинге 7.15.

Листинг 7.15. Объявление типа EquilateralTriangle

```
declare const EquilateralTriangleType: unique symbol;
class EquilateralTriangle extends Triangle {
    [EquilateralTriangleType]: void;
    /* члены класса EquilateralTriangle */
}
```

В качестве упражнения посмотрите, что будет при сочетании типов-сумм с наследованием. Будут ли работать функции `makeShape()`, возвращающая `EquilateralTriangle | Square`, и `draw()`, принимающая в качестве аргумента `Triangle | Square | Circle`? А как насчет `makeShape()`, возвращающей `Triangle | Square`, и `draw()`, принимающей в качестве аргумента `EquilateralTriangle | Square | Circle`?

Работу подобной разновидности подтипизации должен обеспечивать компилятор. Нам не удалось бы добиться подобного поведения в смысле подтипов при «самодельном» типе-сумме, таком как `Variant`, который мы обсуждали в главе 3. Напомню, что `Variant` может служить оберткой для одного из нескольких типов, но сам не является ни одним из этих типов.

7.3.2. Подтипизация и коллекции

Теперь рассмотрим типы, содержащие набор значений некоего другого типа. Начнем с массивов в листинге 7.16. Можно ли передать массив объектов `Triangle` в функцию `draw()`, принимающую в качестве аргумента массив объектов `Shape`, если `Triangle` — подтип `Shape`?

Листинг 7.16. Triangle вместо Shape

```
class Shape {
    /* члены класса Shape */
}

declare const TriangleType: unique symbol;
class Triangle extends Shape {
    [TriangleType]: void;
    /* члены класса Triangle */
}

declare function makeTriangles(): Triangle[];
declare function draw(shapes: Shape[]): void;

draw(makeTriangles());
```

Triangle — подтип Shape

Функция makeTriangles() возвращает массив объектов Triangle

Функция draw() принимает в качестве аргумента массив объектов Shape

Массив объектов Triangle можно использовать вместо массива объектов Shape

Возможно, данное наблюдение не слишком удивительно, однако очень важно: *массивы сохраняют отношение «тип — подтип» типов, из которых состоят*. Как и следовало ожидать, обратный этому код не сработает: если попытаться передать массив объектов `Triangle` вместо ожидаемого массива объектов `Shape`, код просто не скомпилируется (рис. 7.4).

Как мы видели в главе 2, массивы — готовые базовые типы многих языков программирования. А что будет, если описать пользовательскую коллекцию, например `LinkedList<T>` (листинг 7.17)?

Даже в случае непростого типа данных TypeScript правильно определяет, что `LinkedList<Triangle>` — подтип `LinkedList<Shape>`. Как и ранее, противо-

положный вариант не компилируется; передать `LinkedList<Shape>` в качестве `LinkedList<Triangle>` нельзя.

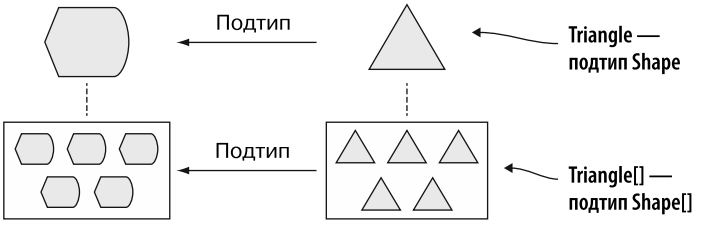


Рис. 7.4. Если Triangle — подтип Shape, то массив объектов Triangle является подтипом массива объектов Shape. Допустимость использования Triangle в качестве Shape позволяет применять массив объектов Triangle в качестве массива объектов Shape

Листинг 7.17. LinkedList<Triangle> вместо LinkedList<Shape>

```
class LinkedList<T> {
    value: T;
    next: LinkedList<T> | undefined = undefined;

    constructor(value: T) {
        this.value = value;
    }

    append(value: T): LinkedList<T> {
        this.next = new LinkedList(value);
        return this.next;
    }
}

declare function makeTriangles(): LinkedList<Triangle>;
declare function draw(shapes: LinkedList<Shape>): void;

draw(makeTriangles());
```

Коллекция — обобщенный связный список

Функция makeTriangles() теперь возвращает связный список объектов Triangle

Функция draw() принимает в качестве аргумента связный список объектов Shape

Код компилируется

КОВАРИАНТНОСТЬ

Тип, сохраняющий отношение «тип — подтип» типа, на основе которого создан, называется ковариантом (covariant). Массивы — ковариантны, поскольку сохраняют отношение «тип — подтип»: Triangle — подтип Shape, поэтому Triangle[] — подтип Shape[].

Различные языки ведут себя по-разному в отношении массивов и коллекций наподобие `LinkedList<T>`. В C#, например, нам пришлось бы описывать интерфейс и использовать ключевое слово `out` (`ILinkedList<out T>`) для явного указания ковариантности таких типов, как `LinkedList<T>`. В противном случае компилятор не смог бы обнаружить отношение «тип — подтип».

В качестве альтернативы ковариантности можно просто игнорировать отношение «тип — подтип» между двумя заданными типами и считать, что между типами `LinkedList<Triangle>` и `LinkedList<Shape>` нет такого отношения (ни один из них не является подтипом другого). Это не касается TypeScript, но имеет смысл в языке C#, где `List<Triangle>` и `List<Shape>` не связаны отношением «тип — подтип».

ИНВАРИАНТНОСТЬ

> Тип, игнорирующий отношения «тип — подтип» лежащего в его основе типа, называется инвариантом (invariant). Тип List<T> языка C# — инвариант, поскольку игнорирует отношение «Triangle — подтип Shape», поэтому List<Triangle> и List<Shape> не связаны в C# отношением «тип — подтип».

Теперь, когда мы обсудили, как связаны друг с другом коллекции в смысле подтипизации, и рассмотрели две распространенные разновидности вариантности, посмотрим, как связаны между собой функциональные типы данных.

7.3.3. Подтипизация и возвращаемые типы функций

Начнем с простого сценария: посмотрим, какие подстановки типов возможны между функцией, возвращающей `Triangle`, и функцией, возвращающей `Shape`, как показано в листинге 7.18. Объявим две функции-фабрики: `makeShape()`, возвращающую объект `Shape`, и `makeTriangle()`, возвращающую объект `Triangle`.

Далее мы реализуем функцию `useFactory()`, принимающую в качестве аргумента функцию типа `() => Shape` и возвращающую объект `Shape`. Попробуем передать в нее `makeTriangle()`.

Листинг 7.18. () => Triangle вместо () => Shape

```
declare function makeTriangle(): Triangle;
declare function makeShape(): Shape;

function useFactory(factory: () => Shape): Shape {
    return factory();
}

let shape1: Shape = useFactory(makeShape);
let shape2: Shape = useFactory(makeTriangle);
```

Функция useFactory() принимает на входе функцию без аргументов, возвращающую объект Shape, и вызывает ее

Обе функции: и makeTriangle(), и makeShape() — можно использовать в качестве аргумента для useFactory()

Ничего необычного в этом коде нет: можно спокойно передать возвращающую `Triangle` функцию вместо функции, возвращающей `Shape`, поскольку возвращаемое значение (`Triangle`) — подтип `Shape`, а значит, его можно присвоить `Shape` (рис. 7.5).

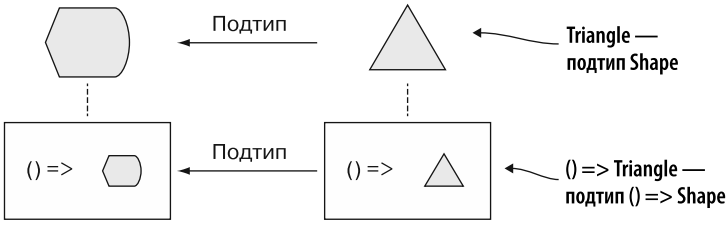


Рис. 7.5. Если Triangle — подтип Shape, то функция, возвращающая Triangle, пригодна к использованию вместо функции, возвращающей Shape, поскольку всегда можно присвоить Triangle вызывающей стороне, где ожидается Shape

Наоборот не получится: если изменить `useFactory()` так, чтобы она ожидала аргумент типа `() => Triangle`, и попытаться передать ей `makeShape()`, как в листинге 7.19, то код не скомпилируется.

Напомню, этот код очень прост: использовать `makeShape()` в качестве функции типа `() => Triangle` нельзя, поскольку `makeShape()` возвращает объект `Shape`. Этот объект может оказаться `Triangle`, но может — и `Square`. Функция `useFactory()` должна возвращать `Triangle`, так что не может вернуть надтип типа `Triangle`. Конечно, она может вернуть подтип, например `EquilateralTriangle`, если передать ей `makeEquilateralTriangle()`.

Листинг 7.19. () => Shape вместо () => Triangle

```
declare function makeTriangle(): Triangle;
declare function makeShape(): Shape;

function useFactory(factory: () => Triangle): Triangle {
    return factory();
}

let shape1: Shape = useFactory(makeShape);
let shape2: Shape = useFactory(makeTriangle);
```

Заменяем здесь Shape на Triangle

Код не компилируется; использовать makeShape() в качестве () => Triangle нельзя

Функции ковариантны относительно возвращаемых типов данных. Другими словами, если `Triangle` — подтип `Shape`, то функциональный тип данных `() => Triangle` окажется подтипом функционального типа `() => Shape`. Обратите внимание: это относится не только к функциональным типам данных, описывающим функции без аргументов. Если и `makeTriangle()`, и `makeShape()` принимают по паре аргументов типа `number`, то все равно будут ковариантны, как мы только что видели.

Функциональные типы данных ведут себя аналогичным образом в большинстве основных языков программирования. Те же правила применимы и для переопределения методов при наследовании типов, при этом изменяется их возвращаемый тип. Если реализовать класс `ShapeMaker`, включающий метод `make()`, который возвращает `Shape`, то можно переопределить его в производном классе `MakeTriangle` так, что он будет возвращать `Triangle`, как показано в листинге 7.20. Компилятор пропустит это, поскольку в результате вызова обоих методов `make()` возвращается объект `Shape`.

Листинг 7.20. Переопределение метода с подтипом в качестве возвращаемого типа

```
class ShapeMaker {
    make(): Shape {
        return new Shape();
    }
}

class TriangleMaker extends ShapeMaker {
    make(): Triangle {
        return new Triangle();
    }
}
```

В классе ShapeMaker описан метод make(), возвращающий объект Shape

Класс TriangleMaker наследует класс ShapeMaker

В классе TriangleMaker метод make() переопределяется, его возвращаемый тип меняется на Triangle

В свою очередь, это допустимо в большинстве основных языков программирования, поскольку функции в них считаются ковариантами относительно возвращаемого типа. Взглянем теперь на функции, типы аргументов которых являются подтипами друг друга.

7.3.4. Подтипизация и функциональные типы аргументов

Вывернем наш пример наизнанку и вместо функций, возвращающих `Shape` и `Triangle`, рассмотрим функции, принимающие соответственно `Shape` и `Triangle` в качестве аргумента. Назовем их `drawShape()` и `drawTriangle()`. Как же соотносятся друг с другом типы `(argument: Shape) => void` и `(argument: Triangle) => void`?

Создадим еще одну функцию, `render()`, принимающую в качестве аргументов объект `Triangle` и функцию типа `(argument: Triangle) => void`, как показано в листинге 7.21. Она просто вызывает заданную функцию, передавая ей заданный `Triangle`.

Листинг 7.21. Функции отрисовки и визуализации

```
declare function drawShape(shape: Shape): void;
declare function drawTriangle(triangle: Triangle): void;

function render(
    triangle: Triangle,
    drawFunc: (argument: Triangle) => void): void {
    drawFunc(triangle);
}
```

Функция drawShape() принимает аргумент типа Shape; drawTriangle() принимает аргумент типа Triangle

Функция render() ожидает на входе объект Triangle и функцию, принимающую Triangle в качестве аргумента

render() просто вызывает переданную ей в качестве аргумента функцию, передавая ей полученный Triangle

А вот и самое интересное: в данном случае можно спокойно передать `drawShape()` в функцию `render()`! Там, где ожидается `(argument: Triangle) => void`, можно использовать `(argument: Shape) => void`.

Это довольно логично: мы передаем объект `Triangle` в функцию отрисовки в качестве аргумента. Если она сама ожидает `Triangle`, как наша функция `drawTriangle()`, то все, конечно, работает. Но схема должна работать и для функции, ожидающей *надтип* типа `Triangle`. Функции `drawShape()` требуется на входе геометрическая фигура — любая — для отрисовки. А поскольку в ней не используется ничего специфического для треугольников, то она более универсальна по сравнению с `drawTriangle()`; она может принимать в качестве аргумента любую геометрическую фигуру: либо `Triangle`, либо `Square`. Поэтому в данном конкретном случае отношение «тип — подтип» меняется на обратное.

КОНТРВАРИАНТНОСТЬ

Тип, который меняет на обратное отношение «тип — подтип» лежащего в его основе типа, называется контрвариантом (contravariant). В большинстве языков программирования функции являются контрвариантами относительно своих аргументов. Вместо ожидающей Triangle в качестве аргумента функции можно подставить функцию, ожидающую в качестве аргумента Shape. Отношение у этих функций будет обратным к отношению типов их аргументов. Если Triangle — подтип Shape, то тип функции, принимающей Triangle в качестве аргумента, будет надтипом функции, принимающей в качестве аргумента Shape (рис. 7.6).

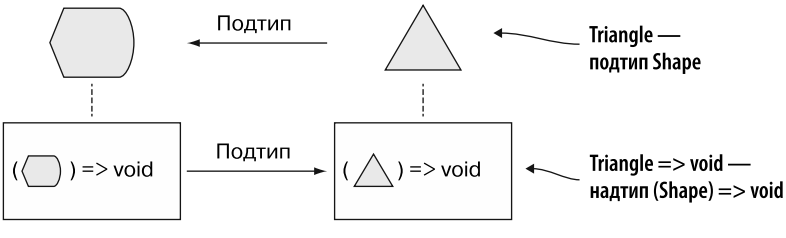


Рис. 7.6. Если Triangle — подтип Shape, то функцию, ожидающую в качестве аргумента Shape, можно использовать вместо функции, ожидающей Triangle, поскольку всегда можно передать Triangle в функцию, принимающую Shape

Ранее я сказал фразу «большинство языков программирования». TypeScript — заметное исключение. В нем возможно и обратное: передать ожидающую подтип функцию вместо ожидающей надтип. Это осознанное архитектурное решение, призванное упростить реализацию распространенных паттернов программирования JavaScript. Впрочем, иногда оно приводит к проблемам во время выполнения.

Рассмотрим пример в листинге 7.22. Для начала опишем в нашем типе `Triangle` метод `isRightAngled()` с целью определить, является ли данный экземпляр прямоугольным треугольником. Реализация данного метода нам неважна.

Теперь рассмотрим обратный пример, приведенный в листинге 7.23. Пусть наша функция `render()` ожидает `Shape` вместо `Triangle` и функцию для отрисовки

произвольных фигур `(argument: Shape) => void` вместо функции, которая умеет рисовать только треугольники `(argument: Triangle) => void`.

Листинг 7.22. Классы Shape и Triangle с методом isRightAngled()

```
class Shape {
    /* члены класса Shape */
}

declare const TriangleType: unique symbol;
class Triangle extends Shape {
    [TriangleType]: void;

    isRightAngled(): boolean {
        let result: boolean = false;

        /* определяем, прямоугольный ли это треугольник */

        return result;
    }

    /* прочие члены класса Triangle */
}
```

Метод isRightAngled() сообщает, описывает ли данный экземпляр прямоугольный треугольник

Листинг 7.23. Модифицированные функции отрисовки и визуализации

```
declare function drawShape(shape: Shape): void;
declare function drawTriangle(triangle: Triangle): void;

function render(
    shape: Shape,
    drawFunc: (argument: Shape) => void): void {
    drawFunc(shape);
}
```

Функции drawShape() и drawTriangle() — такие же, как и ранее

Функция render() ожидает на входе Shape и функцию, принимающую Shape в качестве аргумента

Функция render() просто вызывает указанную функцию, передавая ей полученный объект Shape

А вот как можно вызвать ошибку во время выполнения: описать `drawTriangle()` таким образом, чтобы в ней использовалось нечто специфическое для треугольников, скажем описанный выше метод `isRightAngled()`. А затем вызвать `render()`, передав ей объект `Shape` (а не `Triangle`) и функцию `drawTriangle()`.

Функция `drawTriangle()` в листинге 7.24 получает объект `Shape` и пытается вызвать его метод `isRightAngled()`, но это приведет к ошибке, поскольку `Shape` не `Triangle`.

Данный код компилируется, но вызовет ошибку JavaScript во время выполнения, поскольку среда не сможет обнаружить у объекта `Shape`, переданного функции `drawTriangle()`, метод `isRightAngled()`. Далеко не идеальное поведение, но, как уже упоминалось выше, это осознанное решение создателей языка TypeScript.

В TypeScript, если `Triangle` — подтип `Shape`, то функции типов `(argument: Shape) => void` и `(argument: Triangle) => void` взаимозаменяемы. Фактически они являются подтипами друг друга. Это свойство носит названия *бивариантности* (bivariance).

Листинг 7.24. Попытка вызвать метод isRightAngled() для объекта надтипа типа Triangle

```
function drawTriangle(triangle: Triangle): void {
    console.log(triangle.isRightAngled());
    /* ... */
}

function render(
    shape: Shape,
    drawFunc: (argument: Shape) => void): void {
    drawFunc(shape);
}

render(new Shape(), drawTriangle);
```

Функция drawShape() вызывает для заданного аргумента имеющийся только у Triangle метод

Компилятор позволяет передать объект Shape и метод drawTriangle() для визуализации

БИВАРИАНТНОСТЬ

Два типа бивариантны, если являются подтипами друг друга в случае, когда лежащие в их основе типы находятся в отношении «тип — подтип». В TypeScript если Triangle — подтип Shape, то функциональные типы (argument: Shape) => void и (argument: Triangle) => void являются подтипами друг друга (рис. 7.7).

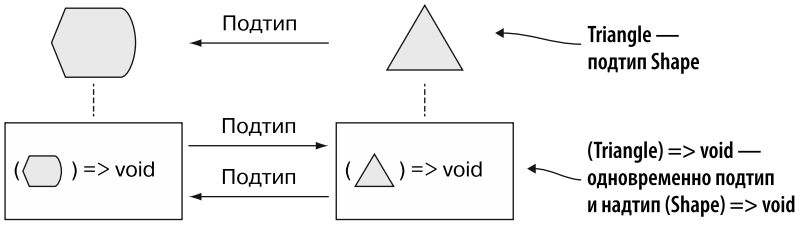


Рис. 7.7. Если Triangle — подтип Shape, то в TypeScript функцию, ожидающую в качестве аргумента Triangle, можно использовать вместо функции, ожидающей Shape. Аналогично и функцию, ожидающую в качестве аргумента Shape, можно применить вместо функции, ожидающей Triangle

Бивариантность функций относительно их аргументов в TypeScript приводит к успешной компиляции неправильного кода. Основная тема данной книги — как исключить благодаря использованию системы типов ошибки выполнения на этапе компиляции. В TypeScript такое осознанное архитектурное решение было принято, чтобы сделать возможными часто встречающиеся паттерны программирования JavaScript.

7.3.5. Краткое резюме по вариантности

В этом разделе мы обсуждали, какие типы могут быть использованы вместо других. Хотя подтипизация проста при обычном наследовании, в случае параметризованных типов данных ситуация усложняется. Среди этих типов — коллекции, функциональные типы и другие обобщенные типы данных. Сохранение, игнорирование,

обращение или превращение в двусторонние отношения «тип — подтип» таких параметризованных типов в зависимости от отношений типов, лежащих в их основе, называется вариантностью.

- Инвариантные типы игнорируют отношение «тип — подтип» лежащих в их основе типов.
- Ковариантные типы сохраняют отношение «тип — подтип» лежащих в их основе типов. Если `Triangle` — подтип `Shape`, то массив типа `Triangle[]` является подтипом массива типа `Shape[]`. В большинстве языков программирования функциональные типы данных ковариантны относительно возвращаемых типов данных.
- Контрвариантные типы обращают отношение «тип — подтип» лежащих в их основе типов. Если `Triangle` — подтип `Shape`, то функциональный тип данных `(argument: Shape) => void` является подтипом функционального типа `(argument: Triangle) => void` в большинстве языков программирования. Но не в TypeScript, в котором функциональные типы данных бивариантны относительно типов их аргументов.
- Бивариантные типы являются подтипами друг друга, если лежащие в их основе типы связаны отношением «тип — подтип». Если `Triangle` — подтип `Shape`, то функциональный тип данных `(argument: Shape) => void` и функциональный тип данных `(argument: Triangle) => void` являются подтипами друг друга (то есть можно взаимозаменить функции обоих типов).

Хотя для разных языков программирования имеются определенные общие правила, не существует *единого способа* поддержки вариантности. Вам нужно понимать, как работает система типов вашего конкретного языка программирования и как она устанавливает отношения «тип — подтип». Это важно, поскольку эти правила определяют, какими типами можно заменять те или иные типы. Нужно ли реализовывать функцию для преобразования `List<Triangle>` в `List<Shape>` или можно просто применить `List<Triangle>` как есть? Ответ зависит от вариантности типа `List<T>` в используемом вами языке программирования.

7.3.6. Упражнения

В следующих упражнениях `Triangle` является подтипом `Shape` и используются правила вариантности языка TypeScript.

1. Можно ли передать переменную типа `Triangle` в функцию `drawShape(shape: Shape) : void`?
2. Можно ли передать переменную типа `Shape` в функцию `drawTriangle(triangle: Triangle) : void`?
3. Можно ли передать массив объектов `Triangle` (`Triangle[]`) в функцию `drawShapes(shape: Shape[]) : void`?

4. Можно ли присвоить функцию `drawShape()` переменной функционального типа данных `(triangle: Triangle) => void`?
5. Можно ли присвоить функцию `drawTriangle()` переменной функционального типа данных `(shape: Shape) => void`?
6. Можно ли присвоить функцию `getShape(): Shape` переменной функционального типа данных `() => Triangle`?

Резюме

- Мы рассмотрели подтипизацию и два способа, с помощью которых в языках программирования определяется, является ли тип подтипом другого типа: структурный и номинальный.
- Мы изучили методику TypeScript, предназначенную для имитации номинальной подтипизации в языке со структурной подтипизацией.
- Мы рассмотрели одно из приложений высшего типа данных, типа, расположенного вверху иерархии типов: безопасную десериализацию.
- Мы рассмотрели также одно из приложений низшего типа данных, типа, расположенного вверху иерархии типов, как тип значения на случай ошибок.
- Мы обсудили также отношения «тип — подтип» между типами-суммами. Тип-сумма, состоящий из меньшего количества типов, является надтипом типа-суммы, состоящего из большего количества типов.
- Мы узнали о существовании ковариантных типов. Массивы и коллекции часто ковариантны, а функциональные типы данных ковариантны относительно возвращаемых типов.
- В некоторых языках программирования типы могут быть инвариантами (не связаны отношением «тип — подтип»), даже если лежащие в их основе типы связаны этим отношением.
- Функциональные типы данных обычно контрвариантны относительно типов аргументов. Другими словами, их отношение «тип — подтип» — противоположное отношению типов их аргументов.
- В TypeScript функции бивариантны относительно типов их аргументов. Если типы аргументов связаны отношением «тип — подтип», то типы функций являются подтипами друг друга.
- В различных языках программирования вариантность реализована по-разному. Важно знать, как устанавливаются отношения «тип — подтип» в используемом вами языке программирования.

Теперь, подробно рассмотрев вопросы подтипизации, мы можем перейти к важнейшей сфере применения подтипизации, о которой мы еще почти не говорили: объектно-ориентированному программированию. В главе 8 мы рассмотрим его элементы и их приложения.

Ответы к упражнениям

7.1. Различаем схожие типы в TypeScript

1. Да — у `Painting` такая же форма, что и у `Wine`, плюс дополнительное свойство `painter`. Вследствие структурной подтипизации в TypeScript `Painting` является подтипом `Wine`.
2. Нет — у типа `Car` отсутствует свойство `name`, описанное в типе `Wine`, так что даже при структурной подтипизации `Car` нельзя использовать вместо `Wine`.

7.2. Присваиваем что угодно, присваиваем чему угодно

1. Да — `never` является подтипом для всех прочих типов данных, включая `number`, так что его можно присвоить переменной типа `number` (хотя мы никогда не сможем создать настоящее значение, поскольку `makeNothing()` никогда ничего не вернет).
2. Нет — `unknown` является надтипом всех прочих типов данных, включая `number`. Можно присвоить `number` переменной типа `unknown`, но не наоборот. Сначала необходимо убедиться, что возвращаемое `makeSomething()` значение является числом, прежде чем присваивать его `x`.

7.3. Допустимые подстановки

1. Да — `Triangle` можно подставить везде, где ожидается `Shape`.
2. Нет — нельзя использовать надтип вместо подтипа.
3. Да — массивы ковариантны, так что массив объектов `Triangle` можно применять вместо массива объектов `Shape`.
4. Да — функции бивариантны относительно аргументов в TypeScript, так что можно использовать `(shape: Shape) => void` вместо `(triangle: Triangle) => void`.
5. Да — функции бивариантны относительно аргументов в TypeScript, так что можно использовать `(triangle: Triangle) => void` вместо `(shape: Shape) => void.`
6. Нет — в TypeScript функции бивариантны относительно аргументов, но не возвращаемых типов данных. Функцию типа `() => Shape` нельзя использовать вместо функции типа `() => Triangle`.

8 Элементы объектно-ориентированного программирования

В этой главе

- Описание контрактов с помощью интерфейсов.
- Реализация иерархии выражений.
- Реализация паттерна проектирования «Адаптер».
- Расширение поведения с помощью примесей.
- Альтернативы чистому объектно-ориентированному программированию.

В данной главе мы рассмотрим элементы объектно-ориентированного программирования и научимся эффективно их применять. Возможно, эти понятия вам уже знакомы, ведь они встречаются во всех объектно-ориентированных языках, так что мы сосредоточимся на конкретных сценариях использования.

Мы начнем с интерфейсов и взглянем на них как на контракты. После интерфейсов мы займемся наследованием: как данных, так и поведения. Альтернативой наследованию является *композиция* (composition). Мы обсудим некоторые различия между этими двумя подходами и узнаем, когда какой из них лучше использовать. Мы поговорим о расширении данных и поведения с помощью *примесей* (mix-ins) или в случае языка TypeScript *типов-пересечений*. Не все языки программирования поддерживают примеси. Не потому, что с объектно-ориентированным программированием неладно, а просто многие программисты считают его единственным подходом к проектированию ПО, и используют его слишком широко.

Но прежде всего я приведу определение объектно-ориентированного программирования.

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

> Объектно-ориентированное программирование — парадигма программирования, в основе которой лежит понятие объекта, содержащего как данные, так и код. Данные определяют состояние объекта. Код состоит из одного или нескольких методов, называемых также сообщениями (messages). В объектно-ориентированной системе объекты могут «общаться» (обмениваться сообщениями) друг с другом с помощью вызовов методов друг друга.

Две ключевые возможности объектно-ориентированного программирования — *инкапсуляция* (encapsulation), позволяющая скрывать данные и методы, и *наследование* (inheritance), дающее возможность расширять тип данными и/или кодом.

8.1. Описание контрактов с помощью интерфейсов

В этом разделе мы попытаемся ответить на часто задаваемый относительно ООП вопрос: в чем разница между абстрактным классом и интерфейсом? Возьмем, к примеру, систему журналирования. Нам нужен метод `log()`, но хотелось бы сохранить возможность использовать и другие реализации журналирования. Это достижимо несколькими способами. Для начала объявим абстрактный класс `ALogger` и создадим несколько наследующих его конкретных реализаций, например `ConsoleLogger`, как показано в листинге 8.1.

Листинг 8.1. Абстрактный механизм журналирования

```
abstract class ALogger {
    abstract log(line: string): void;
}

class ConsoleLogger extends ALogger {
    log(line: string): void {
        console.log(line);
    }
}
```

ALogger — абстрактный класс

log() — абстрактный метод без реализации

Наследующий ALogger класс ConsoleLogger содержит реализацию метода log()

Пользователю нашей системы журналирования можно передавать `ALogger` в качестве параметра. При этом можно передать любой из подтипов `ALogger`, например `ConsoleLogger`, всюду, где ожидается `ALogger`.

В качестве альтернативного варианта можно объявить интерфейс `ILogger` и наследующий его класс `ConsoleLogger`, как показано в листинге 8.2.

Листинг 8.2. Интерфейс для журналирования

```
interface ILogger {
    log(line: string): void;
}

class ConsoleLogger implements ILogger {
    log(line: string): void {
        console.log(line);
    }
}
```

В интерфейсе ILogger объявлен метод log()

Реализующий интерфейс ILogger класс ConsoleLogger содержит метод log()

В таком случае пользователь системы журналирования получает в качестве параметра `ILogger`. При этом можно передать любой реализующий этот интерфейс тип, например `ConsoleLogger`, всюду, где ожидается `ILogger`.

Эти два подхода очень близки и оба вполне работоспособны, но в вышеприведенном сценарии лучше использовать интерфейс, поскольку он задает *контракт*.

ИНТЕРФЕЙСЫ (КОНТРАКТЫ)

Интерфейс (контракт) — это описание набора сообщений, понятных любому реализующему его объекту. Сообщения являются методами и включают название, аргументы и возвращаемый тип данных. У интерфейса нет никакого состояния. Подобно контрактам в реальном мире, которые представляют собой письменные соглашения, интерфейс — это письменное соглашение относительно возможностей, которые будут предоставлять его реализации.

Именно это нам и требуется в данном случае: контракт на журналирование, состоящий из метода `log()`, который смогут вызывать клиенты. Объявление интерфейса `ILogger` ясно демонстрирует всем, кто будет читать наш код, что мы задаем контракт.

Абстрактный класс тоже на это способен, как и на многое другое: он может содержать неабстрактные методы или состояние. Единственное отличие между абстрактным и «обычным» *(конкретным)* классом — невозможность непосредственно создать экземпляр абстрактного класса. Передавая экземпляр абстрактного класса, такой как аргумент типа `ALogger`, на самом деле мы всегда работаем с экземпляром наследующего `ALogger` типа, например `ConsoleLogger`.

Это достаточно тонкое, но важное различие между абстрактными классами и интерфейсами: отношение между `ConsoleLogger` и `ALogger` называется *отношением is-a* (is-a relationship), как, например, `ConsoleLogger` является (is a) `ALogger`, поскольку наследуется от этого абстрактного класса. С другой стороны, от интерфейса `ILogger` ничего не наследуется, ведь он просто описывает контракт. Класс `ConsoleLogger` реализует этот контракт, но при этом семантически не создает отношения *is-a*. Данный класс *удовлетворяет условиям контракта* `ILogger`, но *не является разновидностью* `ILogger`. Поэтому даже в тех языках, где класс может наследоваться только от одного

другого класса, например в Java и C#, классам все равно разрешено реализовывать несколько интерфейсов.

Обратите внимание: интерфейс можно расширить, создав на его основе новый, с дополнительными методами. Например, как демонстрирует листинг 8.3, можно создать интерфейс `IExtendedLogger`, добавляющий в контракт `ILogger` методы `warn()` и `error()`.

Листинг 8.3. Расширенный интерфейс для механизма журналирования

```
interface ILogger {
    log(line: string): void;
}

interface IExtendedLogger extends ILogger {
    warn(line: string): void;
    error(line: string): void;
}
```

Интерфейс IExtendedLogger включает методы log(), warn() и error()

Любой объект, удовлетворяющий условиям контракта `IExtendedLogger`, удовлетворяет также автоматически условиям контракта `ILogger`. Можно также объединить несколько интерфейсов в один. Например, описать интерфейс `ISpeakerWithVolumeControl`, объединяющий два интерфейса, `ISpeaker` и `IVolumeControl`, в один, как показано в листинге 8.4. Это позволит использовать в качестве контракта возможности как динамика, так и регулировки громкости, и одновременно с этим другие типы смогут реализовывать лишь что-то одно (например, регулировку громкости для микрофона).

Конечно, класс `MySpeaker` может реализовывать оба интерфейса `ISpeaker` и `IVolumeControl` вместо `ISpeakerWithVolumeControl`. Однако наличие единого интерфейса позволяет таким компонентам, как `MusicPlayer`, добавить динамик для регулировки громкости. Возможность объединения подобных интерфейсов позволяет создавать их на основе меньших, повторно используемых стандартных блоков.

Именно интерфейсы, а не реализующие их классы, в конечном счете приносят пользу потребителям, так что время, потраченное на поиск оптимальной архитектуры, обычно оказывается оправданным. Известный принцип *программирования интерфейсов* (coding against interfaces) объектно-ориентированного программирования гласит, что следует работать с интерфейсами, а не с классами, как мы делали с `MusicPlayer` в нашем примере. Данный принцип понижает сцепленность компонентов системы, позволяя модифицировать `MySpeaker` или даже заменить его другим типом, никак не повлияв при этом на `MusicPlayer`, если, конечно, удовлетворены условия контракта `ISpeakerWithVolumeContract`.

Задачу привязки конкретной реализации к интерфейсу берут на себя фреймворки внедрения зависимостей, так что остальной код просто запрашивает нужный интерфейс, а фреймворк его предоставляет. За счет этого уменьшается объем «связующего» кода, и мы можем сосредоточить свое внимание на реализации самих компонентов. Мы не будем подробно обсуждать внедрение зависимостей, но это прекрасный подход, позволяющий снизить сцепление кода, особенно удобный для

модульного тестирования, при котором обычно зависимости тестируемых компонентов представляют собой «заглушки» или имитационные объекты.

Листинг 8.4. Объединение интерфейсов

```
interface ISpeaker {
    playSound(/* ... */): void;
}

interface IVolumeControl {
    volumeUp(): void;
    volumeDown(): void;
}

interface ISpeakerWithVolumeControl extends ISpeaker, IVolumeControl {
}

class MySpeaker implements ISpeakerWithVolumeControl {
    playSound(/* ... */): void {
        // конкретная реализация
    }

    volumeUp(): void {
        // конкретная реализация
    }

    volumeDown(): void {
        // конкретная реализация
    }
}

class MusicPlayer {
    speaker: ISpeakerWithVolumeControl;

    constructor(speaker: ISpeakerWithVolumeControl) {
        this.speaker = speaker;
    }
}
```

Интерфейс динамика

Интерфейс регулировки громкости

Объединенный интерфейс динамика и регулировки громкости

Класс MySpeaker реализует этот объединенный интерфейс

Для класса MusicPlayer необходим динамик с возможностями регулировки громкости

Далее мы рассмотрим наследование и некоторые из его приложений.

8.1.1. Упражнения

1. Функция `index()` может использовать экземпляры типов, включающих функцию `getName()`. Как лучше смоделировать этот сценарий?
   А. Объявить конкретный базовый класс `BaseNamed`.
   Б. Объявить абстрактный базовый класс `ANamed`.
   В. Объявить интерфейс `INamed`.
   Г. Проверять во время выполнения, есть ли у данного экземпляра метод `getName()`.

2. В TypeScript в интерфейсе `Iterable<T>` объявлен метод `[Symbol.iterator]`, возвращающий `Iterator<T>`, а в интерфейсе `Iterator<T>` — метод `next()`, возвращающий `IteratorResult<T>`:

```
interface Iterable<T> {
    [Symbol.iterator](): Iterator<T>;
}

interface Iterator<T> {
    next(): IteratorResult<T>;
}
```

   Генераторы возвращают некую смесь того и другого: итерируемый `IterableIterator<T>`, который сам является итератором. Как бы вы описали интерфейс `IterableIterator<T>`?

8.2. Наследование данных и поведения

Наследование — одна из самых известных возможностей объектно-ориентированных языков, позволяющая создавать подклассы родительского класса. Подклассы наследуют как его данные, так и методы. Подкласс, разумеется, является подтипом типа родительского класса, поскольку экземпляр подкласса можно использовать везде, где ожидается родительский.

8.2.1. Эмпирическое правило is-a

А вот сразу же и приложение: при наличии класса, реализующего почти все необходимое нам поведение, можно породить от него другой класс, добавив недостающее. Но если делать это беспорядочно, то проблем только становится вдвое больше. Во-первых, слишком активное использование наследования приводит в итоге к глубоко вложенным иерархиям классов, разобраться и вообще найти что-то в которых очень сложно. Во-вторых, оно приводит к несогласованной модели данных с бессмысленными классами.

Например, если у нас есть класс `Point`, содержащий координаты `x` и `y` точки, то можно унаследовать от него класс `Circle`, добавив свойство `radius`. Круг определяется его центром и радиусом, а `Point` может отражать центр круга. Но это определение выглядит довольно странным (листинг 8.5).

Чтобы понять, почему это выглядит странно, посмотрим на получившееся отношение *is-a*. Является ли экземпляр подкласса логически экземпляром надкласса? В данном случае нет. Круг не разновидность точки. Конечно, при таком определении можно его использовать в данном качестве, но вряд ли найдется обоснованный сценарий, когда это будет уместно.

Листинг 8.5. Пример неудачного наследования

```
class Point {
    x: number;
    y: number;

    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
}

class Circle extends Point {
    radius: number;

    constructor(x: number, y: number, radius: number) {
        super(x, y);
        this.radius = radius;
    }
}
```

Класс Circle наследует координаты x и y своего центра от класса Point

НАСЛЕДОВАНИЕ И ОТНОШЕНИЕ IS-A

Наследование задает отношение is-a между дочерним и родительским типом данных. При базовом классе Shape и дочернем классе Circle образуется отношение «Circle является разновидностью Shape». Оно описывает семантический смысл наследования и позволяет легко проверить, нужно ли использовать наследование для двух заданных типов.

Альтернативный подход — композицию — мы изучим в разделе 8.3. А пока рассмотрим несколько ситуаций, в которых *есть смысл* воспользоваться наследованием.

8.2.2. Моделирование иерархии

Один из случаев, когда стоит применить наследование, — иерархические данные. Это очевидный случай, так что мы не станем обсуждать его подробно. Однако он является оптимальным приложением наследования: при движении вниз по цепи наследования типы данных уточняются, добавляются дополнительные данные и/или поведение (рис. 8.1).

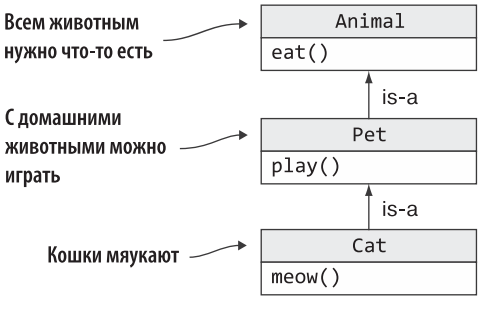


Рис. 8.1. Все животные что-нибудь едят (eat). С домашними животными можно играть (play), но и едят они тоже. А кошки, кроме того, еще и мяукают (meow)

Приведенный на этом рисунке пример может показаться слишком упрощенным, но прекрасно иллюстрирует наследование. Кошка — разновидность домашнего

животного, которое является разновидностью животного, и чем ниже мы спускаемся по иерархии, тем больше вариантов поведения и состояния видим.

Чем выше по иерархии, тем выше уровень абстракции. Если мы хотим просто поиграть (`play()`) с животным, то можем воспользоваться аргументом типа `Pet`. Если же нам нужно, чтобы оно мяукало, то применяем аргумент типа `Cat`.

Это очень простой пример, так что рассмотрим более интересное приложение наследования с хитрым нюансом: несколько производных классов реализуют какое-либо поведение по-разному.

8.2.3. Параметризация поведения выражений

Наследование может пригодиться, еще и когда бóльшая часть поведения и состояния у нескольких типов одинакова и лишь незначительно должна различаться в разных реализациях. Но всем этим типам нужно успешно проходить нашу проверку на *is-a*.

Допустим, у нас есть выражение, результатом вычисления которого является число, бинарные выражения с двумя операндами, а также выражения для суммы и произведения, вычисляемые путем сложения и умножения операндов.

Смоделируем выражение в виде интерфейса `IExpression`, включающего метод `eval()`. Мы сделали его интерфейсом, поскольку никакого состояния он хранить не должен. Далее реализуем абстрактный класс `BinaryExpression` для хранения двух операндов, как показано в листинге 8.6, но оставим метод `eval()` абстрактным, реализовывать его должны будут производные классы. Каждый из классов `SumExpression` и `MulExpression` наследует от `BinaryExpression` два операнда, но включает собственную реализацию метода `eval()` (рис. 8.2).

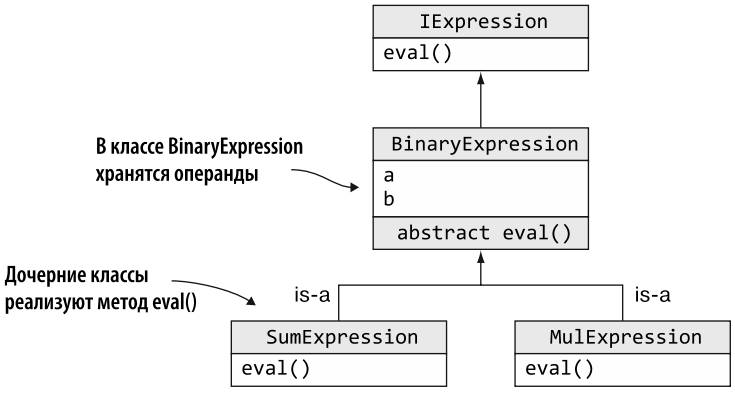


Рис. 8.2. Иерархия выражений с родительским классом BinaryExpression и дочерними классами SumExpression и MulExpression

Эта реализация удовлетворяет нашему критерию *is-a*: `SumExpression` является разновидностью `BinaryExpression`. По мере спуска вниз по иерархии наследуются

общие для классов части (в нашем случае два операнда), но каждый из порожденных классов реализует свой метод `eval()`.

Листинг 8.6. Иерархия выражений

```
interface IExpression {
    eval(): number;
}

abstract class BinaryExpression implements IExpression {
    readonly a: number;
    readonly b: number;

    constructor(a: number, b: number) {
        this.a = a;
        this.b = b;
    }

    abstract eval(): number;
}

class SumExpression extends BinaryExpression {
    eval(): number {
        return this.a + this.b;
    }
}

class MulExpression extends BinaryExpression {
    eval(): number {
        return this.a * this.b;
    }
}
```

Нет нужды делать IExpression классом, поскольку у него нет состояния

BinaryExpression — класс, в котором хранятся два операнда

Метод eval() — абстрактный, поэтому реализации для него не приведено

Оба класса, SumExpression и MulExpression, порождены от класса BinaryExpression и реализуют метод eval()

Следует остерегаться слишком глубоких иерархий классов, усложняющих навигацию по коду, поскольку наследование различных элементов состояния и методов объекта происходит с разных уровней иерархии.

Обычно дочерние классы делают конкретными, а все родительские — абстрактными. Благодаря этому становится проще отслеживать происходящее и избегать непредвиденного поведения. Таковое может возникать, если дочерний класс переопределяет один из методов родительского, а мы затем приводим его экземпляр к родительскому типу и передаем далее в качестве объекта родительского типа. Подобный объект ведет себя не так, как экземпляр родительского класса, что не очевидно для сопровождающих код разработчиков.

В некоторых языках программирования существует способ явным образом отметить дочерний класс как ненаследуемый, чтобы прекратить на нем иерархию наследования. Обычно для этого служат такие ключевые слова, как `final` и `sealed`. Рекомендуется использовать их при любой возможности. Переопределить и расширить поведение позволит лучшая альтернатива, чем наследование, — композиция.

8.2.4. Упражнения

1. Какой из следующих вариантов описывает правильное применение наследования?
   А. `File` (Файл) расширяет `Folder` (Каталог).
   Б. `Triangle` расширяет `Point`.
   В. `Parser` (Средство синтаксического разбора) расширяет `Compiler` (Компилятор).
   Г. Ни один из приведенных выше вариантов.
2. Расширьте описанный в этом разделе пример, добавив в него класс `UnaryExpression` для выражения с одним операндом и класс `UnaryMinusExpression` для обращения знака операнда (скажем, 1 превращается в –1, а –2 превращается в 2).

8.3. Композиция данных и поведения

Один из широко известных принципов объектно-ориентированного программирования гласит, что следует при любой возможности предпочитать композицию наследованию. Посмотрим, что же такое композиция.

Вернемся к нашему примеру с классами `Point` и `Circle`. Мы можем сделать `Circle` дочерним классом `Point`, хотя такое решение будет не вполне правильным. Расширим наш пример, добавив в него класс `Shape`, как показано в листинге 8.7. Допустим, что у всех геометрических фигур в нашей системе должен быть идентификатор, поэтому в классе `Shape` мы опишем свойство `id` типа `string`. `Circle` является разновидностью `Shape`, так что мы можем унаследовать от него `id`. С другой стороны, у круга *есть* центр, вследствие чего у него будет и свойство `center` типа `Point`.

Листинг 8.7. Наследование и композиция

```
class Shape {
    id: string;

    constructor(id: string) {
        this.id = id;
    }
}

class Point {
    x: number;
    y: number;

    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }
}

class Circle extends Shape {
```

Класс Circle наследует от класса Shape свойство id

```
    center: Point;
    radius: number;

    constructor(id: string, center: Point, radius: number) {
        super(id);
        this.center = center;
        this.radius = radius;
    }
}
```

Класс Circle содержит свойство типа Point, описывающее координаты x и y его центра

8.3.1. Эмпирическое правило has-a

Подобно критерию *is-a*, с помощью которого мы проверяли, должен ли класс `Circle` наследовать `Point`, существует и аналогичный критерий для композиции: *has-a* (рис. 8.3).

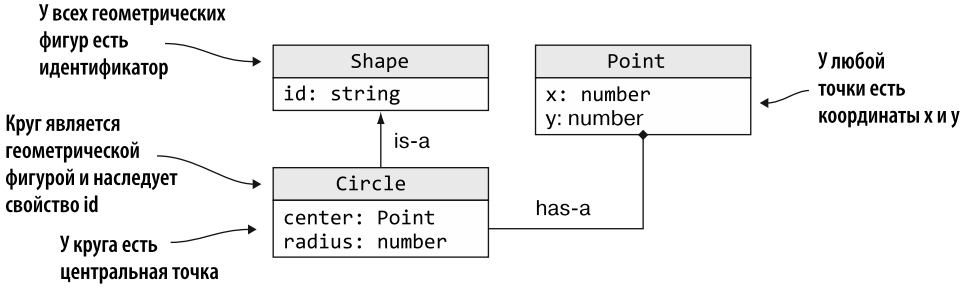


Рис. 8.3. У всех геометрических фигур есть свойство id. Круг является геометрической фигурой, так что наследует id. У круга есть точка, задающая его центр

Вместо того чтобы наследовать поведение от какого-либо типа, можно описать его свойство. Эта методика тоже позволяет хранить состояние нужного типа, но в виде части-компонента типа, а не унаследованной части типа.

КОМПОЗИЦИЯ И ОТНОШЕНИЕ HAS-A

Композиция задает отношение has-a между типом-контейнером и содержащимся в нем типом. Если тип-контейнер — Circle, а содержащийся в нем тип — Point, то их отношение можно описать как «У Circle есть Point» (точка, задающая его центр). Оно описывает семантический смысл композиции и позволяет легко проверить, следует ли использовать композицию для двух заданных типов.

Главное преимущество композиции заключается в следующем: все содержащееся в свойствах-компонентах состояние (например, координаты центра круга) в этих компонентах инкапсулировано, что делает тип данных намного понятнее.

У экземпляра `circle` нашего типа `Circle` имеется свойство `circle.id`, унаследованное им от класса `Shape`, но координаты `x` и `y` его центральной точки хранятся в свойстве `center`: `circle.center.x` и `circle.center.y`. При желании можно сделать

свойство `center` приватным, в результате чего внешний код не будет иметь доступ к нему. Но сделать подобное с унаследованным свойством нельзя: если свойство `id` объявлено в классе `Shape` как публичное, то класс `Circle` не может скрыть его.

Мы рассмотрим далее несколько вариантов применения композиции, но в целом лучше использовать именно ее, а не наследование, чтобы делать состояние и поведение доступными для классов. Композицию имеет смысл задействовать по умолчанию, разве что между двумя типами присутствует четкое отношение *is-a*.

8.3.2. Композитные классы

Мы начнем еще с одного простого, очевидного примера, поскольку опять же с этой концепцией вы, вероятно, хорошо знакомы. Она встречается повсюду в объектно-ориентированном программировании (и не только в нем).

Возьмем для примера компанию с множеством составных частей: различные подразделения, текущий бюджет, генеральный директор (CEO) и т. д. Все они — свойства класса `Company`. Мы обсуждали подобные типы в главе 3, когда говорили о типах-произведениях. Если взглянуть на множество возможных состояний компании, то станет понятно, что оно является декартовым произведением состояний всех подразделений, бюджета, генерального директора и т. д. Дополнительный нюанс: можно инкапсулировать части этого состояния, объявляя их в реализации как приватные свойства и добавив в композитный класс дополнительные методы, которые смогут к ним обращаться в реализации (что недоступно для внешних функций).

Например, нельзя просто подойти к генеральному директору компании и задать ему вопрос. Можно попытаться отправить генеральному директору сообщение через официальные каналы компании, а он уже может ответить или нет, как показывает листинг 8.8.

Листинг 8.8. Вопрос генеральному директору

```
class CEO {
    isBusy(): boolean {
        /* ... */
    }

    answer(question: string): string {
        /* ... */
    }
}

class Department {
    /* ... */
}

class Budget {
    /* ... */
}

class Company {
```

Генеральный директор очень занятой человек и может отвечать или не отвечать на вопросы

В компании есть генеральный директор и несколько подразделений. Есть бюджет

```
    private ceo: CEO = new CEO();
    private departments: Department[] = [];
    private budget: Budget = new Budget();

    askCEO(question: string): string | undefined {
        if (!this.ceo.isBusy()) {
            return this.ceo.answer(question);
        }
    }
}
```

Чтобы задать вопрос генеральному директору, необходимо обратиться к нему через компанию

Если генеральный директор не занят, то ответит нам

Возможность скрыть члены класса и управлять доступом к ним — одно из ключевых отличий инкапсуляции по сравнению с обычными типами-произведениями вроде кортежей и записей.

Типы-значения и ссылочные типы

Возможно, вы слышали о *типах-значениях* (value types) и *ссылочных типах данных* (reference types) либо о различиях между *структурами* и *классами*. Несмотря на множество нюансов, к сожалению, особых обобщений тут сделать не получится. В различных языках программирования данные типы реализуются по-разному, так что вам придется разбираться в нюансах работы именно вашего языка.

В целом в момент присваивания переменной экземпляра типа-значения или передачи его в качестве аргумента функции его содержимое копируется в память, в результате чего фактически создается отдельный экземпляр. В случае же присваивания экземпляра ссылочного типа данных копируется не все состояние, а только ссылка на него. Как старая, так и новая переменная ссылаются на один объект, и через них можно менять его состояние.

Мы не станем здесь подробно говорить на данную тему именно потому, что не хотим запутывать читателя нюансами реализации этих понятий в различных языках программирования. Например, в C# структура очень напоминает класс, но является типом-значением; при ее присваивании копируется состояние. И наоборот, Java не поддерживает настоящих типов-значений, за исключением готовых простых числовых типов данных: все типы представляют собой ссылки. И у C++ тоже есть свои отличия: структура в C++ отличается от класса только тем, что ее члены по умолчанию публичны, а в классе — приватны. В C++ любой тип — значение, если не объявлен явным образом как указатель (*) или ссылка (&). В некоторых функциональных языках программирования используются неизменяемые данные, и разница между значением и ссылкой на него отсутствует вследствие постоянного перемещения данных.

Различие между типами-значениями и типами-ссылками играет важную роль (копирование больших объемов данных отрицательно сказывается на производительности; но лучше копировать, чем использовать совместно, поскольку безопаснее, когда у данных только один владелец). И вам лучше разобраться в этих нюансах применительно к вашему языку программирования.

Далее посмотрим еще на одно, вероятно, не столь очевидное приложение композиции: исключительно полезный паттерн проектирования «Адаптер».

8.3.3. Реализация паттерна проектирования «Адаптер»

С помощью паттерна «Адаптер» можно сделать два класса совместимыми без модификации какого-либо из них. Данный паттерн очень напоминает физический адаптер. Например, возьмем ноутбук, имеющий только порты USB, который нужно подключить к проводной сети, что можно сделать с помощью кабеля Ethernet. Адаптер Ethernet-to-USB осуществляет преобразование между двумя несовместимыми компонентами, Ethernet и USB, и обеспечивает их взаимодействие.

В качестве примера представьте внешнюю библиотеку, которая включает некие необходимые нам геометрические операции, но не вписывается в нашу объектную модель. Она требует описания кругов в соответствии с интерфейсом `ICircle`, в котором объявлено два метода для получения координат `x` и `y` центра круга, `getCenterX()` и `getCenterY()`, а также еще один метод, `getDiameter()`, для получения диаметра круга, как показано в листинге 8.9.

Листинг 8.9. Библиотека геометрических операций

```
namespace GeometryLibrary {

    export interface ICircle {
        getCenterX(): number;
        getCenterY(): number;
        getDiameter(): number;
    }

    /* определенные в интерфейсе ICircle операции */
}
```

Библиотека геометрических операций ожидает, что круги будут соответствовать определенному контракту

Мы не станем описывать здесь сами операции, поскольку для нашего примера они неважны

Наш `Circle` задается своим центром (центральной `Point`) и радиусом. Если класс `Circle` составляет только малую часть нашей огромной кодовой базы, то вряд ли нам захочется проводить ее глобальный рефакторинг лишь для обеспечения совместимости с этой библиотекой. Хорошая новость состоит в том, что есть более простое решение: можно реализовать класс `CircleAdapter` — реализующую необходимый интерфейс обертку для класса `Circle`, содержащую логику преобразования `Circle` в ожидаемый библиотекой вид (листинг 8.10).

Листинг 8.10. Класс CircleAdapter

```
class CircleAdapter implements GeometryLibrary.ICircle {
    private circle: Circle;

    constructor(circle: Circle) {
        this.circle = circle
    }
```

Класс CircleAdapter реализует интерфейс ICircle, ожидаемый библиотекой

CircleAdapter — обертка для экземпляра Circle

```
    getCenterX(): number {
        return this.circle.center.x;
    }

    getCenterY(): number {
        return this.circle.center.y;
    }

    getDiameter(): number {
        return this.circle.radius * 2;
    }
}
```

Методы getCenterX() и getCenterY() служат для получения из объекта Circle соответствующих координат x и y

Метод getDiameter() получает радиус и умножает его на 2 (диаметр равен удвоенному радиусу)

Теперь для того, чтобы использовать библиотеку геометрических операций с экземпляром `Circle`, можно создать для него `CircleAdapter` и передать этот адаптер в библиотеку. Паттерн проектирования «Адаптер» очень удобен для работы с кодом, который мы не можем модифицировать, например кодом из внешних библиотек. Общая структура данного паттерна приведена на рис. 8.4.

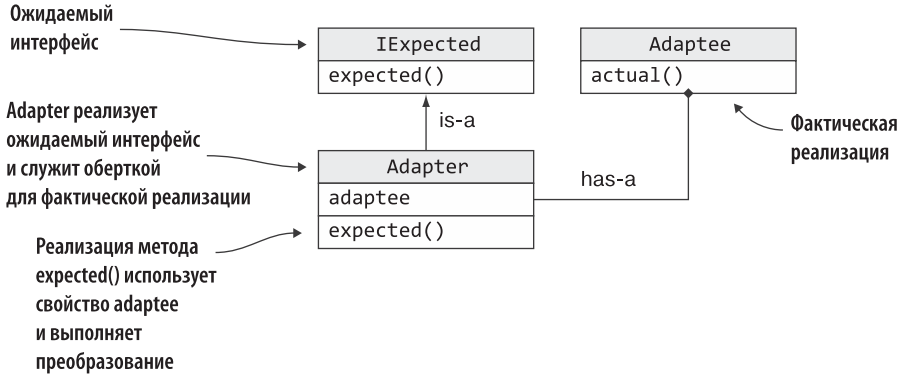


Рис. 8.4. Интерфейс IExpected и фактическая реализация Adaptee несовместимы. Их совместимость обеспечивает Adapter за счет реализации IExpected и преобразования между функционалом, объявленным в IExpected, и функционалом, фактически реализованным в Adaptee

8.3.4. Упражнения

1. Как бы вы смоделировали класс `FileTransfer` (Передача файлов), использующий тип `Connection` (Соединение) для передачи файлов по сети?
   А. Класс `FileTransfer` расширяет `Connection` (наследует необходимое для соединения поведение от типа `Connection`).
   Б. Класс `FileTransfer` реализует `Connection` (реализует интерфейс, в котором объявлено необходимое для соединения поведение).
   В. Класс `FileTransfer` служит адаптером для `Connection` (необходимую для соединения функциональность обеспечивает член класса).

Г. Тип `Connection` расширяет абстрактный класс `FileTransfer` (расширяет абстрактный класс `FileTransfer` и обеспечивает необходимое дополнительное поведение).

2. Реализуйте тип `Airplane` (Самолет) с двумя крыльями и двигателем на каждом из крыльев на основе заданного класса `Engine` (Двигатель). Попробуйте смоделировать этот сценарий с помощью композиции.

8.4. Расширение данных и вариантов поведения

Еще один способ включить дополнительные данные и поведение в тип — нечто отличное от наследования, хотя и, к сожалению, реализуется с помощью наследования в большинстве поддерживающих его языков.

Вернемся к нашему упрощенному примеру с животными: тип `Cat` — разновидность `Pet`, являющегося разновидностью `Animal`. Включим в нашу иерархию тип `WildAnimal` (Дикое животное) и его дочерний тип `Wolf` (Волк). Дикие животные могут бродить (`roam()`), а волк — еще и охотиться. Охота состоит из трех отдельных методов: `track()` (выслеживать), `stalk()`(подкрадываться) и `pounce()` (атаковать) (рис. 8.5).

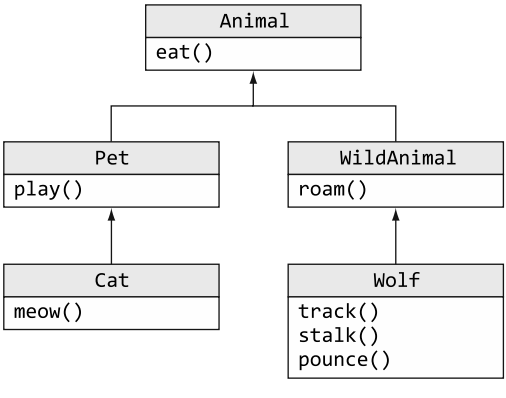


Рис. 8.5. Расширенная иерархия животных с WildAnimal и Wolf. Дикие животные могут бродить (roam()), а волк может охотиться с помощью методов track(), stalk() и pounce()

При желании можно даже реализовать интерфейс `IHunter` (Охотник) со стандартными методами `track()`, `stalk()` и `pounce()`.

А что, если добавить в эту иерархию еще и тип `Tiger`? `Tiger` также может охотиться, и, допустив предположение, что охотничьи повадки хищников одинаковы, нет смысла дублировать код в типах `Wolf` и `Tiger`. Один из вариантов решения этой проблемы — введение в иерархию общего типа `Hunter`, который бы являлся дочерним для `WildAnimal` и родительским для `Wolf` и `Tiger` (рис. 8.6).

Этот подход работает нормально, пока мы не осознаем, что кошки тоже могут охотиться. Как же обеспечить доступ `Cat` к поведению охотника, не прибегая к полной перестройке иерархии типов?

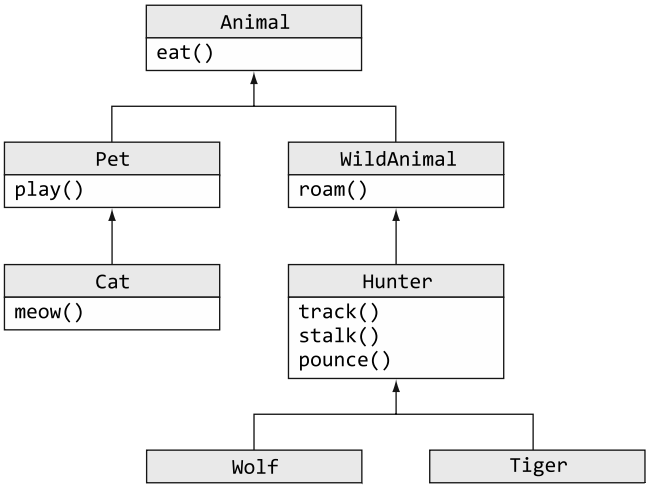


Рис. 8.6. Тип Hunter, описывающий поведение охотника, — родительский для типов Wolf и Tiger

8.4.1. Расширение вариантов поведения с помощью композиции

Один из способов — описать интерфейс `IHunter` и класс `HuntingBehavior`[1], инкапсулирующий общие для кошек, волков и тигров охотничьи повадки, как показано в листинге 8.11. А затем можно будет сделать все наши три типа: `Cat`, `Wolf` и `Tiger` — обертками для экземпляров `HuntingBehavior` и перенести в него реализацию интерфейса (рис. 8.7).

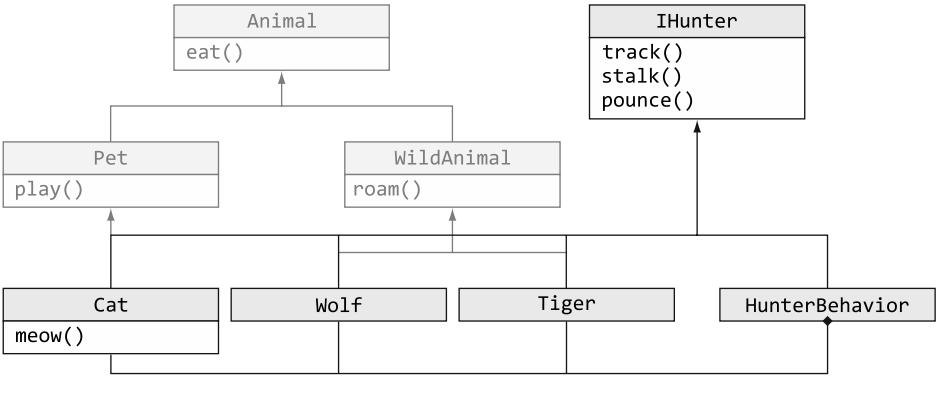


Рис. 8.7. Cat, Wolf и Tiger служат обертками для экземпляра HuntingBehavior и реализуют интерфейс IHunter. Они делегируют все вызовы обернутому объекту. HuntingBehavior предоставляет реализацию IHunter, которую все реализующие его животные могут использовать в виде компонента. Мы исключили HuntingBehavior из иерархии Animal

[1] Автор здесь и далее попеременно использует названия HunterBehavior и HuntingBehavior, хотя речь вроде бы идет об одном классе. — *Примеч. пер.*

Листинг 8.11. Охотничьи повадки

```
interface IHunter {
    track(): void;
    stalk(): void;
    pounce(): void;
}

class HuntingBehavior implements IHunter {
    pray: Animal | undefined;

    track(): void {
        /* ... */
    }

    stalk(): void {
        /* ... */
    }

    pounce(): void {
        /* ... */
    }
}

class Cat extends Pet implements IHunter {
    private huntingBehavior: HuntingBehavior = new HuntingBehavior();

    track(): void {
        this.huntingBehavior.track();
    }

    stalk(): void {
        this.huntingBehavior.track();
    }

    pounce(): void {
        this.huntingBehavior.track();
    }

    meow(): void {
        /* ... */
    }
}
```

Общий интерфейс IHunter

Охотничьи повадки, общие для всех хищных животных

Класс Cat служит оберткой для экземпляра HuntingBehavior

Все методы интерфейса IHunter просто перенаправляются на выполнение экземпляру HuntingBehavior

Это вполне работоспособный подход, но при нем код содержит несколько классов, реализующих интерфейс `IHunter` с помощью обертывания экземпляра `HuntingBehavior`. Добавление каждого нового хищника в нашу иерархию теперь требует массы стереотипного кода, который придется копировать из другого типа. Хуже того, любое изменение интерфейса `IHunter` приведет к каскаду изменений в кодовой базе, ведь придется менять описания всех животных с охотничьими повадками, несмотря на то что меняется только сам `HuntingBehavior`.

Можно ли реализовать это все более оптимально? И да и нет.

8.4.2. Расширение поведения с помощью примесей

Для реализации общего поведения всех хищников проще было бы примешать его во все типы. К сожалению, это обычно осуществляется с помощью множественного наследования, что плохо согласуется с материалом, описанным в начале данной главы при обсуждении эмпирического правила *is-a*. И это еще были описаны далеко не все опасности множественного наследования (но если вам интересно, то можете поискать по ключевым словам *«проблема ромбовидного наследования»* (the diamond inheritance problem)).

Мы взглянем на нашу задачу с точки зрения множественного наследования, создадим класс `Hunter`, реализующий поведение охотника, от которого будут порождены все классы животных-хищников. Тогда `Cat` окажется разновидностью как `Animal`, так и `Hunter`.

С другой стороны, примеси и наследование — не совсем одно и то же. Можно создать класс `HuntingBehavior`, реализующий поведение охотника, а классы животных-хищников будут просто *включать* это поведение.

ПРИМЕСИ И ОТНОШЕНИЕ ВКЛЮЧЕНИЯ

Примеси задают отношение включения (includes) между типом и его типом-примесью. Для класса Cat и типа-примеси HuntingBehavior отношение будет выглядеть как «Cat включает HuntingBehavior». Оно описывает семантический смысл примесей, отличный от семантического смысла отношения *is-a* наследования (рис. 8.8).

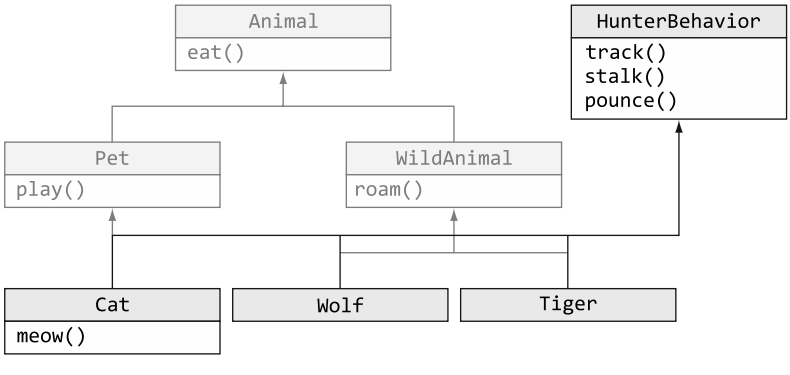


Рис. 8.8. В Cat, Wolf и Tiger примешан класс HuntingBehavior, что позволяет избавиться от большого объема стереотипного кода: необходимости в обертывании классами HuntingBehavior и делегировании вызовов больше нет. Они просто включают это поведение

Примеси столь неоднозначны и противоречивы потому, что многие языки ради упрощения вообще не поддерживают их, а в большинстве из поддерживающих их языков примеси неотличимы от наследования. Все логично, ведь при использовании в качестве примеси такого класса, как `HuntingBehavior`, `Cat` автоматически становится его подтипом. Экземпляр `Cat` можно теперь передать в любое место, где требуется

`HunterBehavior`, но тест на отношение *is-a* ему пройти не удастся: `Cat` не является разновидностью `HunterBehavior`.

Примеси прекрасно помогают уменьшить объем стереотипного кода. Они позволяют сформировать объект, включив в него различные варианты поведения, и использовать повторно общее поведение в различных типах. Лучше всего они подходят для реализации *сквозной функциональности* (cross-cutting concerns): аспектов программы, влияющих на прочую функциональность, которые не получается легко разбить на составные части. Среди них подсчет ссылок, кэширование, сохранение данных и т. д.

Далее мы кратко рассмотрим пример на языке TypeScript, но учтите: его синтаксис специфичен именно для TypeScript. Не беспокойтесь, если он выглядит запутанным, нас интересуют только лежащие в его основе принципы.

8.4.3. Примеси в TypeScript

Чтобы смешать два типа, можно, в частности, воспользоваться функцией `extend()`, которая принимает в качестве аргументов два экземпляра различных типов и копирует все члены второго экземпляра в первый, как показано в листинге 8.12. Я приведу этот пример на языке TypeScript из-за динамической природы лежащего в его основе JavaScript, в котором можно добавлять/удалять члены объекта во время выполнения. Функция `extend()` — обобщенная и может работать с экземплярами любых двух типов.

Листинг 8.12. Расширяем экземпляр типа членами экземпляра другого типа

```
function extend<First, Second>(first: First, second: Second):
    First & Second {
    const result: unknown = {};

    for (const prop in first) {
        if (first.hasOwnProperty(prop)) {
            (<First>result)[prop] = first[prop];
        }
    }
    for (const prop in second) {
        if (second.hasOwnProperty(prop)) {
            (<Second>result)[prop] = second[prop];
        }
    }
    return <First & Second>result;
}
```

Возвращаемый тип функции представляет собой сочетание типов First и Second

Сначала проходим в цикле по всем членам первого объекта и копируем их в переменную result

Далее проделываем то же самое с членами второго типа

В этом листинге нам впервые встречается синтаксис `&`: выражение `First & Second` определяет тип, включающий все члены типов `First` и `Second`. Такой тип в TypeScript называется *типом-пересечением* (intersection type). Не обращайте особого внимания

на эту конкретную реализацию, главное здесь — сама идея объединения двух типов в третий, включающий все их члены.

В большинстве языков программирования не получится так легко добавить новые члены в объект во время выполнения, но это допустимо в JavaScript, а значит, и в TypeScript. В качестве альтернативы, реализуемой на этапе компиляции, в C++ можно воспользоваться множественным наследованием для объявления типа в виде сочетания двух других типов.

Теперь, после описания метода `extend()`, можно модифицировать пример с животными так, как показано в листинге 8.13. Вместо класса `Cat` мы объявим `MeowingPet` — дочерний класс класса `Pet`, представляющий собой животное, умеющее мяукать, но это не совсем `Cat`, ведь у него нет охотничьих повадок. Далее объявим класс `Cat` в виде типа-пересечения `MeowingPet` и `HuntingBehavior`. И при каждом создании нового экземпляра `Cat` будем создавать новый экземпляр `MeowingPet` и расширять (`extend()`) его новым экземпляром `HuntingBehavior`.

Листинг 8.13. Примешиваем поведение

```
class MeowingPet extends Pet {
    meow(): void {
        /* ... */
    }
}

class HunterBehavior {
    track(): void {
        /* ... */
    }

    stalk(): void {
        /* ... */
    }

    pounce(): void {
        /* ... */
    }
}

type Cat = MeowingPet & HunterBehavior;

const fluffy: Cat = extend(new MeowingPet(), new HunterBehavior());
```

Вместо класса Cat объявляем MeowingPet — почти Cat, но не умеющий охотиться

Класс HunterBehavior — такой же, как и в предыдущих примерах

Класс Cat теперь представляет собой тип-пересечение MeowingPet и HunterBehavior

Создаем экземпляр Cat, расширяя тип MeowingPet типом HunterBehavior

Можно обернуть вызов `extend()` в функцию `makeCat()`, чтобы упростить создание объектов `Cat`. В отличие от наследования, примеси позволяют задавать различные типы для разных аспектов поведения, а затем собирать их воедино в окончательный тип. Обычно у каждого из них есть часть своих, присущих именно ему свойств и методов — в нашем случае метод `meow()`, — а также какие-то свойства и методы, единые для нескольких типов, например охотничьи повадки нескольких животных.

Мы уже рассмотрели интерфейсы, наследование, композицию и примеси — основные элементы объектно-ориентированного программирования. Теперь посмотрим на несколько альтернатив чисто объектно-ориентированному коду.

8.4.4. Упражнение

Как бы вы смоделировали пересылку писем и посылок, которые можно отслеживать (с помощью метода `updateStatus()`)?

8.5. Альтернативы чисто объектно-ориентированному коду

Польза от объектно-ориентированного программирования огромна. Возможность создавать компоненты с публичными интерфейсами (скрывая в то же время нюансы реализации), которые могут взаимодействовать друг с другом, — ключ к обработке сложных предметных областей с помощью стратегии «разделяй и властвуй».

Тем не менее существует множество других способов проектирования программного обеспечения, как мы видели в примерах из предыдущих глав, демонстрировавших различные реализации паттернов проектирования, таких как «Стратегия», «Декоратор» и «Посетитель». В некоторых случаях альтернативные варианты обеспечивают лучшее расцепление кода, разбиение на компоненты и повторное использование.

Но эти альтернативные варианты не столь популярны, поскольку многие языки программирования создавались как чисто объектно-ориентированные, без поддержки функциональных или обобщенных типов данных и т. п. Хотя в большинство из них поддержку всего этого со временем добавили, многие программисты до сих пор изучают почти исключительно старые чисто объектно-ориентированные методы. Вкратце рассмотрим несколько возможных альтернатив.

8.5.1. Типы-суммы

Мы уже рассматривали типы-суммы в главе 3, когда искали способ реализовать паттерн проектирования «Посетитель» с помощью типа `Variant` и функции `visit()`. Коротко напомню, как этот код выглядит при использовании ООП и без него.

На сей раз возьмем другой сценарий: простой фреймворк UI. Пользовательский интерфейс состоит из дерева объектов `Panel`, `Label` и `Button`. В одном сценарии `Renderer` будет рисовать эти объекты на экране. Во втором `XmlSerializer` будет сериализовать дерево UI в XML, чтобы сохранить его и загрузить в дальнейшем.

Конечно, можно добавить методы для визуализации и сериализации в каждый из элементов UI, но это не идеальное решение: чтобы добавить новый сценарий, придется вносить изменения во все классы, составляющие UI. В итоге эти классы «знают слишком много» о среде, в которой используются. В качестве альтернативы можно

задействовать паттерн проектирования «Посетитель», чтобы расцепить сценарии с виджетами UI и не давать им информации о способе их применения в приложении, как показано в листинге 8.14.

Листинг 8.14. Создание посетителя с помощью объектно-ориентированного программирования

```
interface IVisitor {
    visitPanel(panel: Panel): void;
    visitLabel(label: Label): void;
    visitButton(button: Button): void;
}

class Renderer implements IVisitor {
    visitPanel(panel: Panel) { /* ... */ }
    visitLabel(label: Label) { /* ... */ }
    visitButton(button: Button) { /* ... */ }
}

class XmlSerializer implements IVisitor {
    visitPanel(panel: Panel) { /* ... */ }
    visitLabel(label: Label) { /* ... */ }
    visitButton(button: Button) { /* ... */ }
}

interface IUIWidget {
    accept(visitor: IVisitor): void;
}

class Panel implements IUIWidget {
    /* члены класса Panel опущены*/
    accept(visitor: IVisitor) {
        visitor.visitPanel(this);
    }
}

class Label implements IUIWidget {
    /* члены класса Label опущены */
    accept(visitor: IVisitor) {
        visitor.visitLabel(this);
    }
}

class Button implements IUIWidget {
    /* члены класса Button опущены */
    accept(visitor: IVisitor) {
        visitor.visitButton(this);
    }
}
```

В объектно-ориентированной реализации для связывания системы воедино необходимы интерфейсы `IVisitor` и `IUIWidget`. Чтобы работать, все виджеты

пользовательского интерфейса должны знать об интерфейсе `IVisitor`, хотя реальная надобность в этом отсутствует.

В альтернативной реализации — с помощью `Variant` (листинг 8.15) — интерфейсы не нужны, как и элементы документа не должны знать о существовании посетителей.

Листинг 8.15. Создание посетителя с помощью вариантного типа данных

```
class Renderer {
    renderPanel(panel: Panel) {/* ... */ }
    renderLabel(label: Label) {/* ... */ }
    renderButton(button: Button) {/* ... */ }
}

class XmlSerializer {
    serializePanel(panel: Panel) {/* ... */ }
    serializeLabel(label: Label) {/* ... */ }
    serializeButton(button: Button) {/* ... */ }
}

class Panel {
    /* члены класса Panel опущены*/
}

class Label {
    /* члены класса Label опущены */
}

class Button {
    /* члены класса Button опущены */
}

let widget: Variant<Panel, Label, Button> =
    Variant.make1(new Panel());

let serializer: XmlSerializer = new XmlSerializer();

visit(widget,
    (panel: Panel) => serializer.serializePanel(panel),
    (label: Label) => serializer.serializeLabel(label),
    (button: Button) => serializer.serializeButton(button)
);
```

Описанный в главе 3 тип Variant способен хранить несвязанные типы данных

Метод visit() «склеивает» систему воедино, подбирая метод сериализации для виджета пользовательского интерфейса

Мы рассмотрели использование типа `Variant` и метода `visit()`, хотя формально лишь первые пять определений классов являются эквивалентом объектно-ориентированного примера. Обратите внимание: никакие интерфейсы тут не нужны.

В целом можно схожим образом передавать объекты различных типов или размещать их в одной коллекции, даже если они не реализуют один интерфейс и не

являются потомками одного родительского типа данных. Вместо этого можно воспользоваться типом-суммой и получить то же поведение, не прибегая к созданию какой-либо связи между типами.

8.5.2. Функциональное программирование

Пока объектно-ориентированные языки программирования не начали поддерживать функциональные типы данных, приходилось обертывать любой элемент поведения в класс. Как мы видели в главе 5, типичная реализация паттерна «Стратегия» требовала наличия интерфейса для описания поведения и нескольких классов, реализующих этот интерфейс.

Снова посмотрим на рисунки из главы 5, в которых описывались две альтернативные реализации паттерна проектирования «Стратегия» (рис. 8.9).

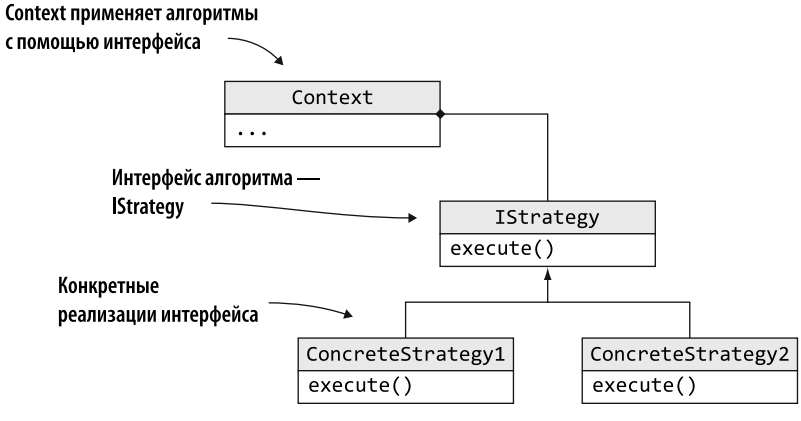


Рис. 8.9. Объектно-ориентированный паттерн проектирования. Различные версии алгоритма реализованы в ConcreteStrategy1 и ConcreteStrategy2

Эту архитектуру можно сильно упростить, если передавать реализацию алгоритма в виде функции. Вместо интерфейса воспользуемся функциональным типом данных, а вместо классов будем задействовать функции (рис. 8.10).

Функциональное программирование позволяет также избежать сохранения состояния: функция может принимать набор аргументов, выполнять какие-либо вычисления и возвращать результат без изменения какого-либо состояния.

Вернемся к нашему примеру с бинарным выражением в листинге 8.16 и взглянем на одну из возможных его функциональных реализаций. Если описать выражение, результатом вычисления которого является число, то можно заменить наш интерфейс `IExpression` на функциональный тип данных `Expression`, который не принимает аргументов и возвращает число. Вместо `SumExpression` можно реализовать фабричную функцию `makeSumExpression()`, которая возвращает для двух заданных чисел замыкание, вычисляющее их сумму. Напомню, что замыкание захватывает

состояние — в данном случае аргументы a и b. То же самое справедливо и для умножения.

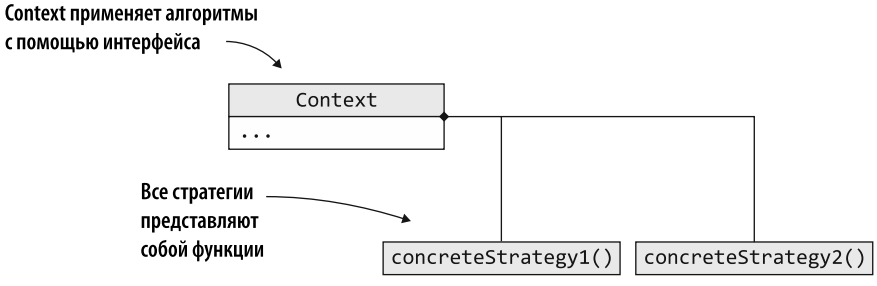


Рис. 8.10. Функциональный паттерн «Стратегия». Различные версии алгоритма реализованы в виде функций

Листинг 8.16. Функциональные выражения

```
type Expression = () => number;

function makeSumExpression(a: number, b: number): Expression {
    return () => a + b;
}

function makeMulExpression(a: number, b: number): Expression {
    return () => a * b;
}
```

Замена для интерфейса IExpression — функциональный тип Expression

Функция makeSumExpression() возвращает замыкание () => a + b

Функция makeMulExpression() возвращает замыкание () => a * b

Класс BinaryExpression нам больше не нужен; мы использовали его для хранения состояния, но состояние теперь обернуто в замыкания.

При более сложном интерфейсе IExpression, имеющем многочисленные методы, объектно-ориентированный подход был бы более оправдан. Но учтите, что в простых случаях можно реализовать то же поведение при гораздо меньшем объеме кода, используя функциональный подход.

8.5.3. Обобщенное программирование

Еще одна альтернатива чисто объектно-ориентированному подходу — обобщенное программирование. Обобщенные типы данных встречались во многих наших примерах кода, но пока не обсуждались подробно. Мы займемся этим в следующих двух главах и рассмотрим различные способы абстрагирования и повторного использования кода.

Пожалуйста, не делайте из данного раздела вывод, что нужно избегать объектно-ориентированного программирования; это инструмент, который играет важную роль при решении широкого спектра задач. Вывод следует сделать другой: надо учитывать несколько возможных альтернатив. Выбирать следует тот подход, который сделает код как можно более безопасным, понятным и слабо сцепленным.

Резюме

- С помощью интерфейсов задаются контракты. Интерфейсы можно расширять и комбинировать.
- Эмпирическое правило *is-a* — отличный критерий того, нужно ли использовать наследование.
- Наследование применяется для отражения иерархий сущностей или реализации параметризованного поведения с помощью абстрактных или переопределенных методов.
- Эмпирическое правило *has-a* — отличный критерий того, когда следует использовать композицию.
- Композиция применяется для инкапсуляции нескольких составных частей в одном типе данных.
- Паттерн проектирования «Адаптер» — пример того, как с помощью инкапсуляции и композиции приспособить тип под другой интерфейс, не модифицируя его.
- С помощью примесей можно добавить в тип дополнительный вариант поведения.
- Типы-суммы, функциональное программирование и обобщенное программирование — заслуживающие внимания альтернативы чистому ООП. Впрочем, они не являются заменой объектно-ориентированного программирования; просто иногда подходят лучше.

Мы лишь вкратце затронули обобщенные типы данных в текущей главе, поскольку две следующие посвящены исключительно этой теме. Читайте дальше!

Ответы к упражнениям

8.1. Описание контрактов с помощью интерфейсов

1. В — с точки зрения функции `index()` это явно контракт, так что лучше будет воспользоваться интерфейсом `INamed`.
2. Этот интерфейс можно задать просто путем сочетания двух остальных интерфейсов:

```
interface IterableIterator<T> extends Iterable<T>, Iterator<T> {
}
```

8.2. Наследование данных и поведения

1. Г — уже по одним названиям классов видно, что ни один из трех примеров не описывает отношения *is-a*, поэтому ни в одном не имеет смысл использовать наследование.

2. Одна из возможных реализаций на основе наследования выглядит так:

```
abstract class UnaryExpression implements IExpression {
    readonly a: number;

    constructor(a: number) {
        this.a = a;
    }

    abstract eval(): number;
}

class UnaryMinusExpression extends UnaryExpression {
    eval(): number {
        return -this.a;
    }
}
```

8.3. Композиция данных и поведения

1. В — этот сценарий отлично подходит для использования композиции. Объект `Connection` следует сделать членом класса `FileTransfer`, поскольку он там необходим, однако не нужно наследовать ни один из этих типов от другого.
2. Одна из возможных реализаций на основе композиции:

```
class Wing {
    readonly engine: Engine = new Engine();
}

class Airplane {
    readonly leftWing: Wing = new Wing();
    readonly rightWing: Wing = new Wing();
}
```

8.4. Расширение данных и вариантов поведения

Один из способов моделирования этого — создать класс `Tracking` для поведения отслеживания, а затем смешать его с классами `Letter` и `Package`, чтобы добавить в них поведение отслеживания. В TypeScript это можно реализовать с помощью такого метода, как `extend()`:

```
class Letter { /*...*/ }
class Package { /*...*/ }

class Tracking {
    setStatus(status: Status) { /*...*/ }
}

type LetterWithTracking = Letter & Tracking;
type PackageWithTracking = Package & Tracking;
```

9 Обобщенные структуры данных

В этой главе

- Разделение независимых элементов функциональности.
- Использование обобщенных структур для размещения данных.
- Обход произвольной структуры данных.
- Формирование конвейера обработки данных.

Мы начнем обсуждение обобщенных типов данных с распространенного сценария их применения: создания независимых, повторно используемых компонентов. Мы рассмотрим несколько сценариев, в которых может пригодиться тождественная функция (функция, просто возвращающая полученный аргумент), и посмотрим на ее обобщенную реализацию. Кроме того, мы обсудим тип `Optional<T>`, который мы создали в главе 3, с точки зрения простого, но обладающего большими возможностями обобщенного типа данных.

Далее мы поговорим о структурах данных. Они определяют форму данных безотносительно к содержанию. Обобщение структур данных позволяет повторно использовать одну форму для самых разнообразных значений, что существенно снижает объем требуемого кода. Мы начнем с бинарного дерева числовых значений и связного списка строк и обобщим их до бинарного дерева и связного списка.

Обобщенные структуры данных — еще не решение всех проблем: необходимо их как-то обходить. Мы обсудим применение итераторов в качестве общего интерфейса для обхода любой структуры данных. Благодаря этому можно также уменьшить

объем требуемого кода, поскольку достаточно будет создать одну работающую с итераторами версию, а не отдельные версии функций для всех структур данных. При этом мы снова воспользуемся генераторами, с которыми мы познакомились в главе 6. Они представляют собой возобновляемые функции, выдающие значения, так что позволяют реализовать итерации по структурам данных.

Наконец, мы поговорим о связывании функций в конвейеры и обработку с их помощью потенциально бесконечных потоков данных.

9.1. Расцепление элементов функциональности

Вы познакомитесь с обобщенными типами на простом примере функции `getNumbers()`, возвращающей массив чисел, позволяя применить к ним нужное преобразование перед возвратом. Для этой цели у нее есть функциональный аргумент `transform()`, который принимает на входе и возвращает число. Вызывающая сторона передает подобную функцию `transform()`, а `getNumbers()` применяет ее, прежде чем вернуть результат, как показано в листинге 9.1.

Листинг 9.1. Функция getNumbers()

```
type TransformFunction = (value: number) => number;

function getNumbers(
    transform: TransformFunction): number[] {
    /* ... */
}
```

Описывающий функцию тип, который принимает на входе и возвращает число

Вызывающая сторона передает функцию transform(), применяемую к каждому из возвращаемых в итоговом массиве чисел

А что, если вызывающая сторона не хочет производить никаких преобразований? Удобным значением по умолчанию для функции `transform()` будет функция, которая ничего не делает — просто возвращает переданное в нее значение, как показано в листинге 9.2.

Листинг 9.2. Функция transform() по умолчанию

```
type TransformFunction = (value: number) => number;

function doNothing(value: number): number {
    return value;
}

function getNumbers(
    transform: TransformFunction = doNothing): number[] {
    /* ... */
}
```

Функция doNothing() просто возвращает переданный ей аргумент без какого-либо его преобразования

Функция doNothing() используется в getNumbers() по умолчанию, поэтому вызывающая сторона может опустить аргумент, если никакого преобразования не требуется

Рассмотрим еще один пример. Допустим, у нас есть массив объектов `Widget` и мы умеем создавать из объектов `Widget` объекты `AssembledWidget`. Функция `assembleWid-`

gets() производит обработку массива объектов Widget и возвращает массив объектов AssembledWidget. А поскольку нам не хотелось бы собирать больше объектов, чем нужно, функция assembleWidgets() принимает в качестве аргумента функцию pluck(), которая возвращает подмножество передаваемого ей массива объектов Widget, как показано в листинге 9.3. Благодаря этому вызывающая сторона может указать функции, какие из виджетов действительно нужны, а какие можно проигнорировать.

Листинг 9.3. Функция assembleWidgets()

```
type PluckFunction = (widgets: Widget[]) => Widget[];

function assembleWidgets(
    pluck: PluckFunction): AssembledWidget[] {
    /* ... */
}
```

Описывающий функцию тип, который возвращает подмножество переданного ей в качестве аргумента массива виджетов

Вызывающая сторона передает функцию pluck(), которую assembleWidgets() вызывает для выбора нужных виджетов

Какое значение по умолчанию следует использовать для функции pluck()? Например, если вызывающая сторона не передала функцию pluck(), то можно преобразовывать весь список виджетов. Будем вызывать по умолчанию в листинге 9.4 эту функцию pluckAll(), которая просто возвращает переданный ей аргумент.

Листинг 9.4. Функция pluck() по умолчанию

```
type PluckFunction = (widgets: Widget[]) => Widget[];

function pluckAll(widgets: Widget[]): Widget[] {
    return widgets;
}

function assembleWidgets(
    pluck: PluckFunction = pluckAll): AssembledWidget[] {
    /* ... */
}
```

Функция pluckAll() просто возвращает весь переданный ей массив

pluckAll() используется в качестве значения по умолчанию для аргумента на случай, если пользователь сам не передаст функцию pluck()

Если сравнить два наших примера, видно, что функции doNothing() и pluckAll() очень похожи: обе принимают аргумент и возвращают его без какой-либо обработки, как показано в листинге 9.5.

Листинг 9.5. Функции doNothing() и pluckAll()

```
function doNothing(value: number): number {
    return value;
}

function pluckAll(widgets: Widget[]): Widget[] {
    return widgets;
}
```

Разница между ними состоит только в типе принимаемого (и возвращаемого) значения: для `doNothing()` это число, а для `pluckAll()` — массив объектов `Widget`. Обе эти функции являются *тождественными* (identity functions). На математическом языке тождественная функция описывается как `f(x) = x`.

9.1.1. Повторно используемая тождественная функция

Нет ничего хорошего в создании двух отдельных функций, которые настолько похожи. Подобный подход очень плохо масштабируется. Можно ли упростить этот процесс, написав повторно используемую тождественную функцию? Да.

Начнем с «наивного» подхода и, поскольку тождественная функция одинаково работает для любого типа, попробуем просто использовать тип `any`. В результате получится функция `identity()`, которая принимает на входе и возращает значение этого типа, как показано в листинге 9.6.

Листинг 9.6. «Наивная» тождественная функция

```
function identity(value: any): any {
    return value;
}
```

Проблема данной реализации заключается в том, что при использовании `any` мы обходим проверку типов и утрачиваем типобезопасность, как показано в листинге 9.7. Результат вызова `identity()` со строкой в качестве аргумента можно спокойно передать функции, ожидающей число, и код скомпилируется без ошибок, но вызовет сбой на этапе выполнения.

Листинг 9.7. Небезопасное использование типа any

```
function square(x: number): number {
    return x * x;
}

square(identity("Hello!"));
```

Оператор скомпилируется и вызовет сбой на этапе выполнения, поскольку обходит обычные проверки типов

Можно реализовать вышеописанное более безопасно: параметризовать то, что в функциях различается, а именно тип аргумента. Такой параметр называется типом-параметром.

ТИП-ПАРАМЕТР

> Тип-параметр (type parameter) — идентификатор названия обобщенного типа. Типы-параметры служат «заполнителями», заменяющими конкретные типы, которые клиент указывает при создании экземпляра обобщенного типа данных.

В листинге 9.8 в нашей обобщенной тождественной функции используется тип-параметр `T` — `number` в первом случае и `Widget[]` во втором.

Листинг 9.8. Обобщенная тождественная функция

```
function identity<T>(value: T): T {
    return value;
}

function getNumbers(
    transform: TransformFunction = identity): number[] {
    /* ... */
}

function assembleWidgets(
    pluck: PluckFunction = identity): AssembledWidget[] {
    /* ... */
}
```

Обобщенная тождественная функция с типом-параметром T

Можно воспользоваться функцией identity() вместо doNothing(). Тип-параметр T в этом случае превращается в number

Можно применить функцию identity() вместо pluckAll(). Тип-параметр T в таком случае превращается в Widget[]

Компилятор достаточно умен и о том, каким должен быть тип `T`, догадается без подсказок. Нам больше не нужны функции `doNothing()` и `pluckAll()`, и эту тождественную функцию можно повторно использовать для любого другого типа. А после определения типа (например, в случае `getNumbers()` тип `T` — `number`) компилятор может проверять типы, и ситуация, когда мы пытаемся возвести в квадрат строку, становится невозможной, как показано в листинге 9.9.

Листинг 9.9. Типобезопасность

```
function identity<T>(value: T): T {
    return value;
}

square(identity("Hello!"));
```

Теперь не компилируется

Эта реализация стала возможной, поскольку внутреннее устройство тождественной функции одно и то же, независимо от типа, для которого она используется. Фактически мы расцепили логику тождественности с предметной областью задач `getNumbers()` и `assembleWidgets()`, поскольку логика тождественности и предметная область задачи *ортогональны* (независимы).

9.1.2. Тип данных Optional

В качестве еще одного примера рассмотрим реализацию типа `Optional` из главы 3 (листинг 9.10). Напомню, что опциональный тип данных может содержать значение какого-то типа `T` или не содержать ничего.

Логика обработки отсутствия значения опять же не зависит от фактического типа значения. Мы используем обобщенный тип данных `Optional`, который может хранить любой тип, поскольку его внутренняя логика обработки от этого не меняется. Тип `Optional` можно рассматривать как совершенно отдельное от типа измерение, поскольку любые изменения в `Optional` не влияют на `T` и, наоборот, никакие изменения `T` не влияют на `Optional`. Подобная изоляция — поистине замечательная возможность обобщенного программирования.

Листинг 9.10. Тип данных Optional

```
class Optional<T> {
    private value: T | undefined;
    private assigned: boolean;

    constructor(value?: T) {
        if (value) {
            this.value = value;
            this.assigned = true;
        } else {
            this.value = undefined;
            this.assigned = false;
        }
    }
    hasValue(): boolean {
        return this.assigned;
    }

    getValue(): T {
        if (!this.assigned) throw Error();

        return <T>this.value;
    }
}
```

Опционал служит адаптером для обобщенного типа данных T

Аргумент value — необязательный, поскольку TypeScript не поддерживает перегрузки конструкторов

Если аргумент value не задан, то при попытке получить значение генерируем исключение

9.1.3. Обобщенные типы данных

Мы только что рассмотрели два сценария использования обобщенных типов данных: обобщенную функцию и обобщенный класс. Теперь вернемся назад и выясним, что же делает обобщенные типы данных особенными. Мы начали эту книгу с обсуждения базовых типов данных и способов их сочетания. Мы рассмотрели такие типы, как `boolean` и `number`, а также `boolean | number`. Далее рассмотрели функциональные типы данных, например `() => number`. Как видим, ни у одного из этих типов нет типа-параметра. Число — это просто число. Функция, возвращающая число, — просто функция, возвращающая число.

С появлением обобщенных типов данных все меняется. Возьмем, например, обобщенную функцию `(value: T) => T` с типом-параметром `T`. Конкретные функции создаются при указании фактического типа `T`. Например, при `Widget[]` в качестве `T` получается функциональный тип `(value: Widget[]) => Widget[]`. Мы впервые можем подключать типы и получать различные описания типов (рис. 9.1).

ОБОБЩЕННЫЕ ТИПЫ ДАННЫХ

Обобщенный тип данных (generic type) — обобщенная функция, класс, интерфейс и т. д., параметризованный по одному или нескольким типам. Благодаря обобщенным типам данных можно писать универсальный код, работающий с различными типами, и добиться высокой степени повторного использования кода.

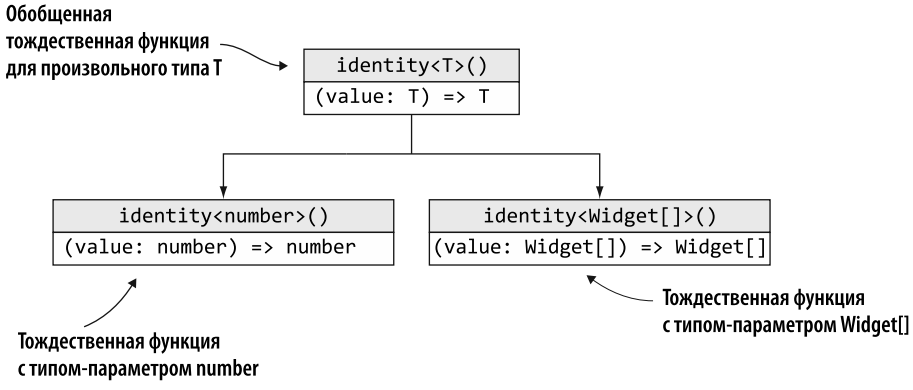


Рис. 9.1. Обобщенная тождественная функция с типом-параметром T и два ее экземпляра: identity<number>() с конкретным типом (value: number) => number и identity<Widget[]>() с конкретным типом (value: Widget[]) => Widget[]

Как мы видели в предыдущих примерах и как увидим далее в этой и следующей главах, при использовании обобщенных типов данных наш код гораздо лучше разбивается на компоненты. Эти обобщенные компоненты можно применять в качестве стандартных блоков и, сочетая их, добиваться желаемого поведения при минимальной их взаимной зависимости. Выйдем за рамки простых примеров `identity<T>()` и `Optional<T>` и рассмотрим структуры данных.

9.1.4. Упражнения

1. Реализуйте обобщенный тип `Box<T>` — простую обертку для значения типа `T`.
2. Реализуйте обобщенную функцию `unbox<T>()`, которая принимает в качестве аргумента экземпляр `Box<T>` и возвращает содержащееся в нем значение.

9.2. Обобщенное размещение данных

Начнем с пары необобщенных примеров: бинарного дерева чисел, приведенного в листинге 9.11, и связного списка строк, показанного в листинге 9.12. Вы наверняка знакомы с этими простыми структурами данных. Мы реализуем дерево в виде одного или нескольких узлов, каждый из которых содержит числовое значение и ссылки на левый и правый дочерние узлы. Причем это могут быть как ссылки на узлы, так и `undefined`, если соответствующий дочерний узел отсутствует.

Листинг 9.11. Бинарное дерево чисел

```
class NumberBinaryTreeNode {
    value: number;
    left: NumberBinaryTreeNode | undefined;
    right: NumberBinaryTreeNode | undefined;
```

```
    constructor(value: number) {
        this.value = value;
    }
}
```

Аналогично мы реализуем связный список в виде одного или нескольких узлов, каждый из которых содержит `string` и ссылку на следующий узел или `undefined`, если такового не существует, как показано в листинге 9.12.

Листинг 9.12. Связный список строк

```
class StringLinkedListNode {
    value: string;
    next: StringLinkedListNode | undefined;

    constructor(value: string) {
        this.value = value;
    }
}
```

Теперь представьте, что в другой части проекта нам понадобилось бинарное дерево строк. Можно реализовать идентичный `NumberBinaryTreeNode` тип `StringBinaryTreeNode`, заменив тип значения с `number` на `string`. Заманчивая перспектива: всего лишь скопировать/вставить код и заменить пару мелочей, но копирование/вставка кода — плохая идея. Представьте, что наш класс включает вдобавок еще и несколько методов. Если мы скопируем эти методы, а потом обнаружим ошибку в одной из версий, то можем забыть исправить ошибку в скопированном варианте. Наверняка вы понимаете, к чему я веду: вместо дублирования кода можно воспользоваться обобщенными типами данных!

9.2.1. Обобщенные структуры данных

Реализуем обобщенное дерево `BinaryTreeNode<T>`, подходящее для хранения любого типа данных, как показано в листинге 9.13.

Листинг 9.13. Обобщенное бинарное дерево

```
class BinaryTreeNode<T> {
    value: T;
    left: BinaryTreeNode<T> | undefined;
    right: BinaryTreeNode<T> | undefined;

    constructor(value: T) {
        this.value = value;
    }
}
```

BinaryTreeNode<T> служит для хранения значения типа T

На самом деле можно не ждать, пока от нас потребуют бинарное дерево строк: сцепление структуры данных бинарное дерево с типом `number` в нашей исходной реализации `NumberBinaryTreeNode` было излишним и ненужным. Аналогичным образом

можно заменить `StringLinkedListNode` на обобщенный список `LinkedListNode<T>`, как показано в листинге 9.14.

Листинг 9.14. Обобщенный связный список

```
class LinkedListNode<T> {
    value: T;
    next: LinkedListNode<T> | undefined;

    constructor(value: T) {
        this.value = value;
    }
}
```

Помните: в большинстве языков программирования уже есть библиотеки, включающие все нужные структуры данных (списки, очереди, стеки, множества, словари и т. д.). Мы рассматриваем их реализации, чтобы продемонстрировать использование обобщенных типов данных, но лучше вообще не писать код. Если есть возможность выбрать готовую обобщенную структуру данных из библиотеки, то следует сделать это.

9.2.2. Что такое структура данных

Немного пофилософствуем и зададимся вопросом: «Что вообще такое структура данных?» Она состоит из трех частей.

- *Сами данные* — значения `number` и `string` в наших деревьях и списках в предыдущем примере. Структуры данных содержат данные.
- *Форма данных* — в нашем бинарном дереве данные расположены иерархически, у каждого элемента есть от нуля до двух дочерних элементов. В нашем списке данные расположены последовательно, элементы идут один за другим.
- *Набор сохраняющих форму данных операций* — например, структура данных может включать набор операций для добавления или удаления элемента. Мы не приводили подобных операций в предыдущих примерах, но понятно желание, чтобы связный список после удаления элемента из его середины, например, остался по-прежнему связным.

Мы видим тут два отдельных элемента функциональности. Один из них — это данные: тип данных и фактически хранящееся в экземпляре структуры данных значение. Второй — форма данных и сохраняющие форму операции. С помощью обобщенных структур данных наподобие тех, которые мы видели в начале этого раздела, можно расцепить эти элементы функциональности. Обобщенные структуры данных отвечают за размещение данных, их форму и все сохраняющие форму операции. Бинарное дерево — это бинарное дерево вне зависимости от того, содержит оно строки или числа. Делегирование ответственности за размещение данных обобщенным структурам данных, не зависящих от фактически хранимых данных, позволяет разбить код на компоненты.

А теперь, считая, что у нас есть все эти структуры данных, посмотрим, как их обходить и просматривать содержимое.

9.2.3. Упражнения

1. Реализуйте структуру данных `Stack<T>` для стека («последним вошел, первым вышел») с методами `push()`, `pop()` и `peek()`.
2. Реализуйте структуру данных `Pair<T, U>` с членами `first` и `second` двух своих типов-параметров соответственно.

9.3. Обход произвольной структуры данных

Допустим, нам нужно обойти наше бинарное дерево по порядку и вывести значения всех его элементов, как показано в листинге 9.15. Напомню, что центрированный обход (in-order traversal, LNR) дерева — это рекурсивный обход в порядке левый — корневой — правый узел (рис. 9.2).

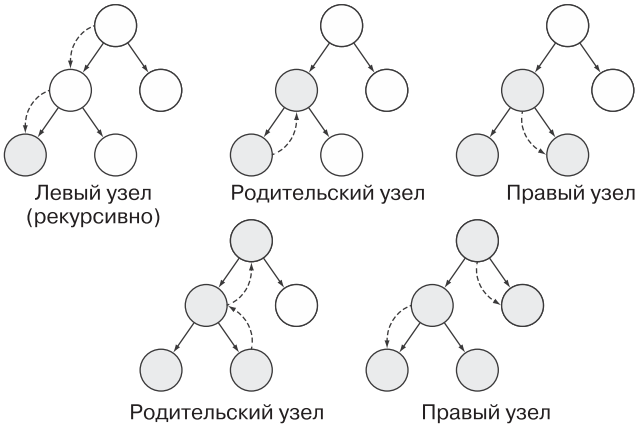


Рис. 9.2. Центрированный обход дерева. Рекурсивно обходим левые узлы, пока не достигнем самого крайнего слева, переходим к его родительскому узлу, а затем к правому узлу этого родительского узла. Далее возвращаемся к родительскому узлу данного родительского узла, а затем переходим к его правому узлу. Идем всегда налево; а затем, когда обойдем все поддерево, переходим к родительскому узлу; после этого идем направо

Листинг 9.15. Вывод по порядку

```
class BinaryTreeNode<T> {
    value: T;
    left: BinaryTreeNode<T> | undefined;
    right: BinaryTreeNode<T> | undefined;

    constructor(value: T) {
        this.value = value;
    }
}
```

То же самое бинарное дерево, что и раньше

```
function printInOrder<T>(root: BinaryTreeNode<T>): void {
    if (root.left != undefined) {
        printInOrder(root.left);
    }

    console.log(root.value);

    if (root.right != undefined) {
        printInOrder(root.right);
    }
}
```

Рекурсивно переходим к левому дочернему узлу, если таковой есть

Выводим значение этого узла

Наконец рекурсивно переходим к правому дочернему узлу, если таковой есть

В качестве примера создадим дерево из нескольких узлов и посмотрим на результаты работы функции `printInOrder()` в листинге 9.16.

Листинг 9.16. Пример работы функции printInOrder()

```
let root: BinaryTreeNode<number> = new BinaryTreeNode(1);
root.left = new BinaryTreeNode(2);
root.left.right = new BinaryTreeNode(3);
root.right = new BinaryTreeNode(4);

printInOrder(root);
```

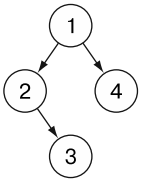


Рис. 9.3. Пример бинарного дерева

Этот код создает приведенное на рис. 9.3 дерево.

Центрированный его обход приводит к такому выводу:

```
2
3
1
4
```

Но что, если нам нужно вывести еще и все значения связного списка строк? Можно реализовать функцию `printList()`, которая обходит список от головы к хвосту и выводит значения всех элементов, как показано в листинге 9.17.

Листинг 9.17. Вывод связного списка

```
class LinkedListNode<T> {
    value: T;
    next: LinkedListNode<T> | undefined;

    constructor(value: T) {
        this.value = value;
    }
}

function printLinkedList<T>(head: LinkedListNode<T>): void {
    let current: LinkedListNode<T> | undefined = head;

    while (current) {
        console.log(current.value);
        current = current.next;
    }
}
```

Та же самая реализация связного списка, что и раньше

Начинаем с головы списка

Повторяем вычисления, пока остаются узлы

Выводим значение узла и переходим к следующему

В качестве конкретного примера можем инициализировать список строк и вывести его с помощью функции `printLinkedList()`, как показано в листинге 9.18.

Листинг 9.18. Пример работы функции printLinkedList()

```
let head: LinkedListNode<string> = new LinkedListNode("Hello");
head.next = new LinkedListNode("World");
head.next.next = new LinkedListNode("!!!");

printLinkedList(head);
```

В результате работы этого кода создается список, приведенный на рис. 9.4.

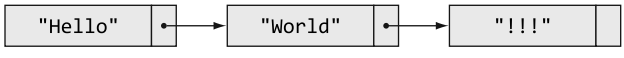


Рис. 9.4. Пример связного списка

В результате запуска этого кода выводится:

```
Hello
World
!!!
```

Этот код работает, но, возможно, существует лучший вариант.

9.3.1. Использование итераторов

Можно ли еще больше разбить код по обязанностям? Обе наши функции, `printInOrder()` и `printLinkedList()`, выполняют две задачи: обход структуры данных и вывод ее содержимого в консоль. И что еще хуже: вторые задачи совпадают, обе функции выводят значения в консоль.

Можно провести небольшое обобщение — вынести обход структуры данных в отдельный компонент. Начнем с бинарного дерева. Наша задача: обойти все элементы дерева по порядку и вернуть значение каждого из узлов. Такой обход мы будем называть *итерацией*; то есть мы производим итеративный обход структуры данных.

ИТЕРАТОР

> Итератор (iterator) — объект, обеспечивающий обход структуры данных. Он предоставляет стандартный интерфейс, скрывающий от клиентов фактическую форму структуры данных.

Реализуем нужные итераторы. Начнем с описания `IteratorResult<T>`, приведенного в листинге 9.19, в виде типа с двумя свойствами: свойством `value` типа `T` и свойством `done` типа `boolean`, указывающего на то, достигли ли мы конца структуры данных.

Листинг 9.19. Тип IteratorResult

```
type IteratorResult<T> = {
    done: boolean;
    value: T;
}
```

В листинге 9.20 описан интерфейс `Iterator<T>`, в котором объявлен единственный метод `next()`, возвращающий `IteratorResult<T>`.

Листинг 9.20. Интерфейс Iterator

```
interface Iterator<T> {
    next(): IteratorResult<T>;
}
```

Теперь можно реализовать `BinaryTreeNodeIterator<T>` в виде класса, реализующего интерфейс `Iterator<T>`, как показано в листинге 9.21. В классе проводится центрированный обход с помощью приватного метода `inOrder()`, а все значения узлов помещаются в очередь. Метод `next()` удаляет из очереди значения благодаря использованию метода `shift()` массива и возвращает значения `IteratorResult<T>` до тех пор, пока есть что возвращать (рис. 9.5).

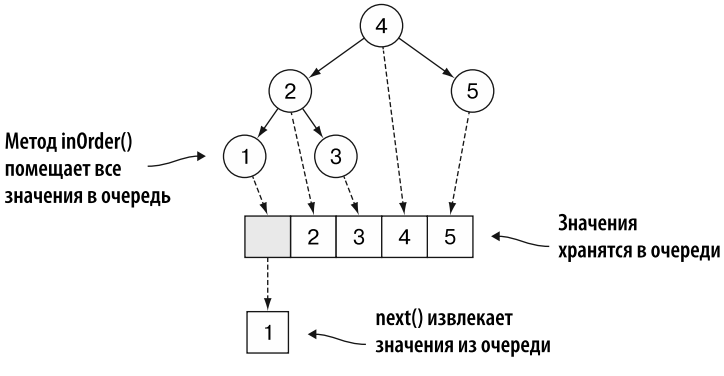


Рис. 9.5. Метод inOrder() обходит узлы бинарного дерева по порядку, добавляя все значения в очередь. Метод next() во время обхода удаляет значения из очереди и возвращает их

Листинг 9.21. Итератор для бинарного дерева

```
class BinaryTreeIterator<T> implements Iterator<T> {
    private values: T[];
    private root: BinaryTreeNode<T>;

    constructor(root: BinaryTreeNode<T>) {
        this.values = [];
        this.root = root;

        this.inOrder(root);
```

Очередь значений

Конструктор выполняет центрированный обход, заполняя очередь значений

```
    }

    next(): IteratorResult<T> {
        const result: T | undefined = this.values.shift();

        if (!result) {
            return {done: true, value: this.root.value };
        }

        return {done: false, value: result };
    }

    private inOrder(node: BinaryTreeNode<T>): void {
        if (node.left != undefined) {
            this.inOrder(node.left);
        }

        this.values.push(node.value);

        if (node.right != undefined) {
            this.inOrder(node.right);
        }
    }
}
```

При каждом вызове метода next() значение извлекается из очереди с помощью вызова shift()

Если результат равен undefined, то присваиваем свойству done значение true и возвращаем какое-то значение по умолчанию

Метод inOrder() обходит узлы по порядку

Добавляем значение каждого из узлов в очередь значений

Это не самая эффективная реализация, поскольку для нее необходима очередь, количество элементов которой совпадает с количеством узлов дерева. Возможен и более эффективный, требующий меньше памяти способ обхода, но его логика сложнее. Пока используем такой пример, поскольку вскоре нам предстоит увидеть более простой и оптимальный способ реализации.

Реализуем также обобщенный класс `LinkedListIterator<T>` для обхода связного списка (листинг 9.22).

Листинг 9.22. Итератор для связного списка

```
class LinkedListIterator<T> implements Iterator<T> {
    private head: LinkedListNode<T>;
    private current: LinkedListNode<T> | undefined;

    constructor(head: LinkedListNode<T>) {
        this.head = head;
        this.current = head;
    }

    next(): IteratorResult<T> {
        if (!this.current) {
            return {done: true, value: this.head.value };
        }

        const result: T = this.current.value;
        this.current = this.current.next;
        return {done: false, value: result };
    }
}
```

Если мы достигли конца списка и свойство current равно undefined, то присваиваем свойству done значение true и возвращаем некое фиктивное значение (которое никогда не должно использоваться)

В переменной result хранится значение текущего узла

Делаем текущим следующий узел списка

Возвращаем сохраненный результат

Разберемся, в чем удобство этих итераторов. Нам больше не нужны отдельные функции для вывода значений всех узлов бинарного дерева и всех строк связного списка. Достаточно одной общей функции, принимающей в качестве аргумента итератор и извлекающей с его помощью выводимые значения, как показано в листинге 9.23.

Листинг 9.23. Функция print(), использующая итератор

```
function print<T>(iterator: Iterator<T>): void {
    let result: IteratorResult<T> = iterator.next();

    while (!result.done) {
        console.log(result.value);
        result = iterator.next();
    }
}
```

Инициализируем переменную result с помощью вызова метода next(), извлекая первое значение

Пока result.done не равно true, мы можем вывести значение и передвинуть итератор

Функция print() — обобщенная, принимающая в качестве аргумента итератор

Поскольку функция `print()` работает с итераторами, можно передать ей как `BinaryTreeIterator<T>`, так и `LinkedListIterator<T>`. Фактически ее можно использовать для вывода в консоль любой структуры данных, если есть итератор, умеющий обходить такую структуру.

Итераторы позволяют повторно использовать гораздо больше кода. Например, чтобы узнать, содержится ли в структуре данных определенное значение, не нужно реализовывать отдельную функцию для каждой структуры данных; можно просто реализовать функцию `contains()`, принимающую на входе итератор и искомое значение, как показано в листинге 9.24, и затем применить ее с любым итератором, реализующим интерфейс `Iterator<T>` (рис. 9.6).

Листинг 9.24. Использующая итератор функция contains()

```
function contains<T>(value: T, iterator: Iterator<T>): boolean {
    let result: IteratorResult<T> = iterator.next();

    while (!result.done) {
        if (result.value == value) return true;
        result = iterator.next();
    }

    return false;
}
```

Итераторы — связующее звено между структурами данных и алгоритмами, позволяющее их расцеплять. Такой подход позволяет сочетать и комбинировать различные структуры данных с различными функциями, лишь бы интерфейс между ними был `Iterator<T>`.

Обратите внимание: обходить многие структуры данных можно различными способами. Мы сосредоточили наше внимание на центрированном обходе бинарного дерева, но существуют также алгоритмы прямого обхода (pre-order traversal, NLR) и обратного обхода (post-order traversal, LRN). Все эти алгоритмы можно

реализовать в виде итераторов по одному и тому же бинарному дереву. Одной структуре данных не обязана соответствовать ровно одна стратегия обхода.

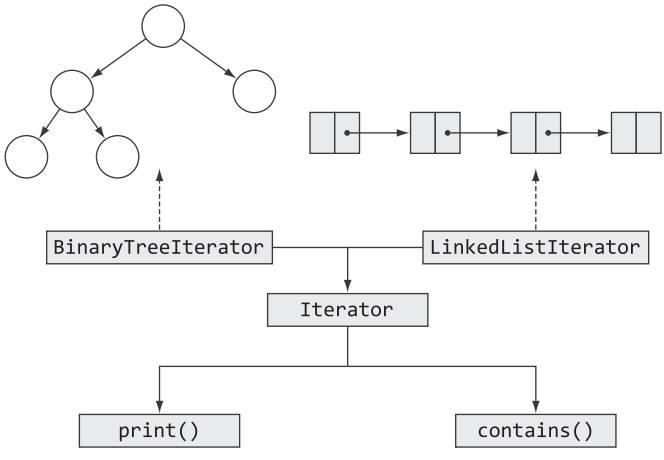


Рис. 9.6. Класс BinaryTreeIterator<T> реализует алгоритм обхода бинарного дерева. Класс LinkedListIterator<T> реализует алгоритм обхода связного списка. Оба класса воплощают контракт Iterator. Функции print() и contains() принимают в качестве аргумента Iterator, так что можно сочетать и комбинировать эти функции с различными структурами данных

9.3.2. Делаем код итераций потоковым

Итераторы — настолько удобный инструмент, что их поддержка появилась в библиотеках большинства основных языков, а в некоторых — даже специальный синтаксис. Мы вкратце касались этого вопроса в главе 6, когда говорили о генераторах, и обсудим его здесь очень подробно.

На самом деле вовсе не нужно описывать типы `IteratorResult<T>` и `Iterator<T>`; в TypeScript уже есть готовые. Эквивалентный интерфейс в C# называется `IEnumerator<T>` и тоже позволяет производить обход структур данных. Эквивалент в Java называется `Iterator<T>`. В библиотеке C++ есть несколько разновидностей итераторов. Мы поговорим об этом подробнее в главе 10 при обсуждении видов итераторов. Ключевым моментом здесь является то, что данный паттерн настолько удобен, что поддерживается «из коробки».

Помимо итераторов, реализующих код обхода структуры данных, есть и другой интерфейс, с помощью которого можно пометить тип как итерируемый: `Iterable<T>` (листинг 9.25).

Листинг 9.25. Интерфейс Iterable

```
interface Iterable<T> {
    [Symbol.iterator](): Iterator<T>;
}
```

Синтаксис `[Symbol.iterator]` — особенность языка TypeScript. Это просто особое название, очень напоминающее трюк с символом, с помощью которого мы реализовывали номинальную подтипизацию во всей нашей книге. В интерфейсе `Iterable<T>` объявлен метод `[Symbol.iterator]()`, возвращающий `Iterator<T>`.

Модифицируем наш тип `LinkedListNode<T>`, сделав его итерируемым (листинг 9.26).

Листинг 9.26. Итерируемый связный список

```
class LinkedListNode<T> implements Iterable<T> {
    value: T;
    next: LinkedListNode<T> | undefined;

    constructor(value: T) {
        this.value = value;
    }

    [Symbol.iterator](): Iterator<T> {
        return new LinkedListIterator<T>(this);
    }
}
```

Мы реализовали интерфейс Iterable<T>, создав для этого списка новый экземпляр LinkedListIterator

Можно также пометить наше бинарное дерево как итерируемое, предоставив аналогичный метод `[Symbol.iterator]` для создания экземпляра `BinaryTreeIterator<T>`.

Итераторы позволяют использовать в TypeScript синтаксис `for ... of`. Это специальный синтаксис для прохода в цикле по всем элементам итерируемого объекта, благодаря которому код становится намного понятнее. Эквиваленты существуют в большинстве основных языков программирования. В C# это `IEnumerable<T>`, `IEnumerator<T>` и циклы `foreach`. В Java — `Iterable<T>`, `Iterator<T>` и циклы `for :`.

Посмотрим еще раз на реализации функций `print()` и `contains()` в листинге 9.27, а затем изменим их, воспользовавшись итерируемыми объектами и циклом `for ... of`.

Листинг 9.27. Функции print() и contains() с аргументом Iterator

```
function print<T>(iterator: Iterator<T>): void {
    let result: IteratorResult<T> = iterator.next();

    while (!result.done) {
        console.log(result.value);
        result = iterator.next();
    }
}

function contains<T>(value: T, iterator: Iterator<T>): boolean {
    let result: IteratorResult<T> = iterator.next();
```

```
    while (!result.done) {
        if (result.value == value) return true;

        result = iterator.next();
    }

    return false;
}
```

А теперь изменим эти функции в листинге 9.28 так, чтобы они принимали аргумент типа `Iterable<T>` вместо `Iterator<T>`. Из `Iterable<T>` всегда можно получить `Iterator<T>`, вызвав метод `[Symbol.iterator]`.

Листинг 9.28. Функции print() и contains() с аргументом Iterable

```
function print<T>(iterable: Iterable<T>): void {
    for (const item of iterable) {
        console.log(item);
    }
}

function contains<T>(value: T, iterable: Iterable<T>): boolean {
    for (const item of iterable) {
        if (item == value) return true;
    }

    return false;
}
```

Для вывода каждого из элементов в консоль функция print() использует цикл for...of

Для сравнения каждого из элементов с заданным значением функция contains() использует цикл for...of

Как можно видеть, код значительно сократился. Вместо прохода в цикле по структурам данных вручную, с помощью `Iterator<T>` и метода `next()`, нам теперь достаточно одной строки кода с циклом `for ... of`.

Теперь посмотрим, как упростить код итератора. Я уже упоминал, что центрированный обход бинарного дерева неэффективен, поскольку все узлы заносятся в очередь перед возвратом их значений. В более эффективном решении обход дерева должен производиться без занесения всех узлов в очередь, но реализация при этом усложнится. В листинге 9.29 приведена предыдущая реализация.

Листинг 9.29. Итератор для бинарного дерева

```
class BinaryTreeIterator<T> implements Iterator<T> {
    private values: T[];
    private root: BinaryTreeNode<T>;

    constructor(root: BinaryTreeNode<T>) {
        this.values = [];
        this.root = root;
        this.inOrder(root);
    }

    next(): IteratorResult<T> {
        const result: T | undefined = this.values.shift();
```

```
        if (!result) {
            return { done: true, value: this.root.value };
        }

        return { done: false, value: result };
    }

    private inOrder(node: BinaryTreeNode<T>): void {
        if (node.left != undefined) {
            this.inOrder(node.left);
        }

        this.values.push(node.value);

        if (node.right != undefined) {
            this.inOrder(node.right);
        }
    }
}
```

Данный код можно заменить на генератор (я уже упоминал генераторы в главе 6). Генератор — это возобновляемая функция, выдающая значения с помощью оператора `yield`, которая при повторном вызове возобновляет выполнение с того места, на котором остановилась. Генераторы в TypeScript возвращают `IterableIterator<T>`, представляющий собой просто сочетание двух уже знакомых нам интерфейсов: `Iterable<T>` и `Iterator<T>`. По реализующему оба эти интерфейса объекту можно пройти в цикле вручную, с помощью метода `next()`, но он применим и в цикле `for ... of`.

Реализуем обход бинарного дерева в виде генератора в листинге 9.30. С помощью генераторов можно реализовать рекурсивный обход и выдавать значения до тех пор, пока не обойдем всю структуру данных (рис. 9.7).

Получилась намного более компактная реализация. Обратите внимание на рекурсивность функции `inOrderIterator()`. На каждом уровне значения выдаются «наружу», пока не дойдут до первоначальной вызывающей стороны.

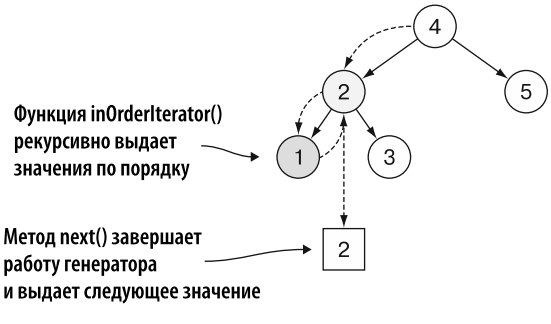


Рис. 9.7. Функция inOrderIterator() — генератор и возвращает IterableIterator<T>. Подобно функции inOrder(), она рекурсивно обходит дерево, но вместо того, чтобы заносить элементы в очередь, выдает их. Вызов метода next() для возвращенного итератора приводит к завершению работы генератора и выдаче следующего значения

Листинг 9.30. Итерация по бинарному дереву с помощью генератора

```
function* inOrderIterator<T>(root: BinaryTreeNode<T>):
    IterableIterator<T> {
    if (root.left) {
        for (const value of inOrderIterator(root.left)) {
            yield value;
        }
    }

    yield root.value;

    if (root.right) {
        for (const value of inOrderIterator(root.right)) {
            yield value;
        }
    }
}
```

function* означает, что функция является генератором, поэтому может выдавать значения, а затем снова возобновлять выполнение

Прежде всего обходим левое поддерево и выдаем все возвращенные значения

Затем выдаем текущее значение

Далее обходим правое поддерево и выдаем все возвращенные значения

Аналогично можно организовать обход связного списка с помощью генератора, значительно упростив логику. Наша первоначальная реализация выглядела так, как показано в листинге 9.31.

Листинг 9.31. Итератор для связного списка

```
class LinkedListIterator<T> implements Iterator<T> {
    private head: LinkedListNode<T>;
    private current: LinkedListNode<T> | undefined;

    constructor(head: LinkedListNode<T>) {
        this.head = head;
        this.current = head;
    }

    next(): IteratorResult<T> {
        if (!this.current) {
            return { done: true, value: this.head.value };
        }

        const result: T = this.current.value;
        this.current = this.current.next;
        return { done: false, value: result };
    }
}
```

Можно заменить этот код еще одним генератором, выдающим значения по мере обхода списка, как показано в листинге 9.32.

Компилятор преобразует этот код в итератор, выдающий значения типа `IteratorResult<T>` при каждом вызове `yield`. Когда функция достигает конца списка и завершает работу (без выдачи значения), возвращается итоговый `IteratorResult<T>` с равным `true` полем `done`.

Листинг 9.32. Итерация по связному списку с помощью генератора

```
function* linkedListIterator<T>(head: LinkedListNode<T>):
    IterableIterator<T> {
    let current: LinkedListNode<T> | undefined = head;

    while (current) {
        yield current.value;
        current = current.next;
    }
}
```

Выдаем значения по мере обхода связного списка

Осталось только включить эти генераторы в сами структуры данных в виде реализаций `[Symbol.iterator]()`. Посмотрим, как выглядит наша итоговая версия связного списка (листинг 9.33).

Листинг 9.33. Итерируемый связный список с использованием генератора

```
class LinkedListNode<T> implements Iterable<T> {
    value: T;
    next: LinkedListNode<T> | undefined;

    constructor(value: T) {
        this.value = value;
    }

    [Symbol.iterator](): Iterator<T> {
        return linkedListIterator(this);
    }
}
```

[Symbol.iterator]() просто возвращает результат linkedListIterator()

Этот код работает, поскольку генератор возвращает `IterableIterator<T>`. Иногда для встраивания вызова генератора внутрь цикла `for...of` (например, `for (const value of linkedListIterator(...))` требуется `Iterable<T>`. А иногда, напротив, нужен `Iterator<T>`, как в предыдущем примере, чтобы использовать цикл `for...of` для экземпляра самой структуры данных.

9.3.3. Краткое резюме по итераторам

Мы начали обсуждение итераторов с пары обобщенных структур данных, определяющих форму данных вне зависимости от их сущности. Мы видели возможности этой абстракции. Но если писать код для обхода каждой структуры данных всякий раз, когда понадобится применить к ней какую-либо операцию наподобие `print()` или `contains()`, то у нас окажется множество версий каждой функции.

Это приводит к появлению интерфейса `Iterator<T>`, расцепляющего форму данных с функциями за счет универсального интерфейса обхода с помощью метода `next()`. Благодаря этому интерфейсу достаточно написать одну версию `print()` и одну версию `contains()`, работающие с итераторами.

Впрочем, обход в цикле путем вызова `next()` с проверкой значения `done` — достаточно неуклюжий способ. На помощь приходит интерфейс `Iterable<T>`, в ко-

тором описывается метод `[Symbol.iterator]()`. Этот метод позволяет получить итератор. Но что еще лучше, можно использовать `Iterable<T>` в операторе `for...of`. Это не просто повышает понятность синтаксиса, но и исключает необходимость работы с итератором явным образом, поскольку при каждой итерации цикла мы получаем сам элемент.

Наконец, мы видели, что можно упростить код обхода за счет использования генератора, выдающего значения по мере обхода структуры данных. Генераторы возвращают `IterableIterator<T>`, так что можно как применить их непосредственно внутри циклов `for...of`, так и реализовать интерфейс `Iterable<T>` для структуры данных.

Как упоминалось ранее, аналогичный специализированный тип, позволяющий использовать цикл `for` для обхода по элементам структуры данных, существует в большинстве основных языков программирования. Что же касается генераторов, то, хоть в языке Java и нет встроенного оператора `yield`, C# поддерживает их с помощью очень близкого к TypeScript синтаксиса.

В общем, описывайте структуры данных так, чтобы они обязательно реализовывали `Iterable<T>`. Избегайте написания функций, включающих обход какой-либо конкретной структуры данных; желательно, чтобы они работали с итераторами и можно было повторно использовать одну и ту же логику для различных структур данных. Обдумайте, не применить ли при реализации логики обхода структуры данных оператор `yield`, ведь он обычно делает код лаконичнее и понятнее.

Усовершенствование IteratorResult<T>

Очень жаль, что нам пришлось использовать `IteratorResult<T>` в качестве возвращаемого типа метода `next()`. Именно так описан данный готовый интерфейс в TypeScript, хоть это и противоречит приведенному в главе 3 принципу, согласно которому лучше возвращать из функции результат или ошибку, а не и то и другое. Тип `IteratorResult<T>` включает булево свойство `done` и свойство `value` типа `T`. По завершении обхода итератором всего списка он возвращает `done`, равное `true`, но должен вернуть и что-то в качестве `value`. Поскольку `value` — обязательная часть результата, необходимо вернуть некое значение по умолчанию, но структура данных уже полностью пройдена. Вызывающий код ни в коем случае не должен использовать `value`, если `done` равно `true`. К сожалению, гарантировать это нельзя.

В качестве лучшего контракта можно предложить тип-сумму, например `Optional<T>` или `T | undefined`. В этом случае можно возвращать объекты `T` до тех пор, пока значения не исчерпаются, а затем, по завершении обхода, не возвращать ничего.

9.3.4. Упражнения

1. Реализуйте прямой обход обобщенного бинарного дерева. При прямом обходе сначала обходится родительский узел, затем левое поддерево, а далее правое. Попытайтесь реализовать его с помощью генератора.
2. Реализуйте функцию, проходящую в цикле по массиву в обратном порядке (с конца в начало).

9.4. Потоковая обработка данных

В этом последнем разделе мы рассмотрим очень интересный аспект итераторов: то, что они могут не быть конечными. В листинге 9.34 мы реализуем функцию, генерирующую бесконечный поток случайных чисел. Мы назовем ее `generateRandomNumbers()`, и она будет выдавать эти числа в бесконечном цикле.

Листинг 9.34. Бесконечный поток случайных чисел

```
function* generateRandomNumbers(): IterableIterator<number> {
    while (true) {                          ← Бесконечный цикл
        yield Math.random();  ← Выдает случайное число
    }                                       на каждом шаге
}
```

Мы вызовем эту функцию сначала для получения `IterableIterator<T>`, а затем вызовем несколько раз `next()` для получения случайных чисел, как показано в листинге 9.35.

Листинг 9.35. Потребление значений из потока данных

```
let iter: IterableIterator<number> = generateRandomNumbers();

console.log(iter.next().value);
console.log(iter.next().value);
console.log(iter.next().value);
```

На практике встречается множество примеров бесконечных потоков данных: чтение символов с клавиатуры, получение данных через сетевое соединение, сбор данных от датчиков и т. д. Для обработки подобных данных можно использовать конвейеры.

9.4.1. Конвейеры обработки

Конвейеры обработки состоят из компонентов — функций, которые принимают в качестве аргумента итератор, производят определенную обработку и возвращают также итератор. Подобные функции можно сцепить для обработки данных по мере поступления. Этот паттерн, лежащий в основе реактивного программирования, широко распространен в функциональных языках.

В качестве примера реализуем функцию `square()`, возводящую в квадрат все числа полученного на входе итератора. Это легко можно сделать с помощью генератора, который принимает в качестве аргумента `Iterable<number>` и выдает квадраты его значений, как показано в листинге 9.36. Обратите внимание: на входе ему достаточно `Iterable<number>` вместо `IterableIterator<number>`, но последний в качестве аргумента тоже подходит, ведь `IterableIterator<number>` также удовлетворяет контракту `Iterable<number>`.

В конвейерах обработки часто применяется функция `take()`, которая возвращает первые `n` элементов полученного на входе итератора, отбрасывая все остальные, как показано в листинге 9.37.

Листинг 9.36. Функция square()

```
function* square(iter: Iterable<number>):
    IterableIterator<number> {
    for (const value of iter) {
        yield value ** 2;
    }
}
```

Эта функция принимает в качестве аргумента Iterable<number> и возвращает IterableIterator<number>

Листинг 9.37. Функция take()

```
function* take<T>(iter: Iterable<T>, n: number):
    IterableIterator<T> {
    for (const value of iter) {
        if (n-- <= 0) return;

        yield value;
    }
}
```

Уменьшаем значение n на единицу и прекращаем работу после выдачи n значений

Выдаем одно значение

Теперь в листинге 9.38 создадим конвейер для возведения в квадрат чисел из бесконечного потока данных и вывода в консоль первых пяти результатов (рис. 9.8).

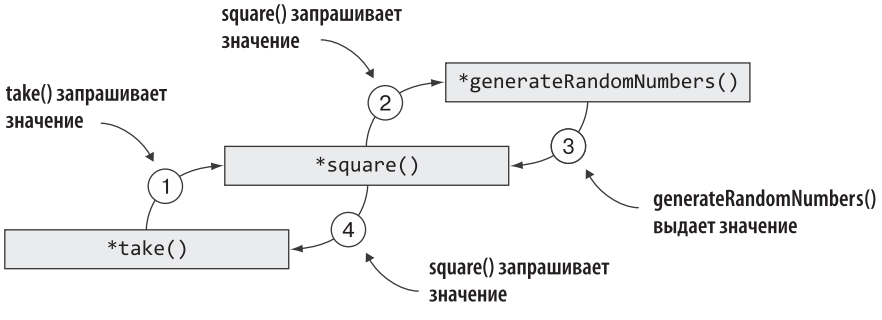


Рис. 9.8. Конвейер и последовательность вызовов. Функция take() запрашивает значение из итератора square(). Тот запрашивает значение из итератора generateRandomNumbers(). Он выдает значение в функцию square(), а та выдает значение в функцию take()

Листинг 9.38. Конвейер

```
const values: IterableIterator<number> =
    take(square(generateRandomNumbers()), 5);

for (const value of values) {
    console.log(value);
}
```

Функция take() получает пять значений из функции square(), которая получает значения из функции generateRandomNumbers()

Итераторы — ключ к созданию подобных конвейеров и к последовательной обработке значений. Важно также четко понимать, что вычисления в подобных конвейерах производятся отложенным образом. В нашем примере переменная `values` — типа `IterableIterator<number>`. И хотя она создается с помощью вызова конвейера,

никакой код пока не выполняется. Значения начинают идти по конвейеру только после начала потребления значений в цикле `for...of`.

В каждой итерации цикла вызывается метод `next()` итератора `values`, который, в свою очередь, вызывает функцию `take()`. Ей для работы нужно значение, поэтому она вызывает `square()`. Той тоже требуется значение для возведения в квадрат, так что она вызывает `generateRandomNumbers()`. Та, в свою очередь, выдает случайное значение функции `square()`, которая возводит его в квадрат и выдает функции `take()`. А она выдает это значение в цикл, где то выводится в консоль.

А поскольку вычисление конвейеров производится отложенным образом, появляется возможность работы с бесконечными генераторами наподобие `generateRandomNumbers()`. Мы обсудим алгоритмы подробнее в главе 10.

9.4.2. Упражнения

1. Еще одна распространенная функция — `drop()`. Это противоположность функции `take()`, она отбрасывает первые `n` элементов итератора и возвращает остальные. Реализуйте `drop()`.
2. Создайте конвейер, который возвращает шестой, седьмой, восьмой, девятый и десятый элементы заданного итератора. Подсказка: это можно сделать с помощью сочетания функций `drop()` и `take()`.

Резюме

- Обобщенные типы данных удобны для разделения независимых элементов функциональности.
- Обобщенные структуры данных отвечают за форму данных вне зависимости от их содержимого.
- Итераторы обеспечивают общий интерфейс для обхода структур данных.
- Тип `Iterator<T>` обозначает итератор, а `Iterable<T>` — сущность, обход которой можно произвести с помощью итератора.
- Итераторы можно реализовать, задействовав генераторы.
- В большинстве языков программирования существуют итераторы и специальный синтаксис для обхода их в цикле.
- Итераторы не обязаны быть конечными: они могут генерировать значения бесконечно.
- С помощью функций, которые принимают на входе и возвращают итераторы, можно формировать конвейеры обработки.

Теперь, после описания обобщенных структур данных, мы можем рассмотреть в главе 10 еще один важнейший ингредиент программирования: алгоритмы.

Ответы к упражнениям

9.1. Расцепление элементов функциональности

1. Одна из возможных реализаций:

```
class Box<T> {
    readonly value: T;

    constructor(value: T) {
        this.value = value;
    }
}
```

2. Одна из возможных реализаций:

```
function unbox<T>(boxed: Box<T>): T {
    return boxed.value;
}
```

9.2. Обобщенное размещение данных

1. Одна из возможных реализаций на основе массива (в JavaScript у массивов есть готовые методы `pop()` и `push()`):

```
class Stack<T> {
    private values: T[] = [];

    public push(value: T) {
        this.values.push(value);
    }

    public pop(): T {
        if (this.values.length == 0) throw Error();

        return this.values.pop();
    }

    public peek(): T {
        if (this.values.length == 0) throw Error();

        return this.values[this.values.length - 1];
    }
}
```

2. Одна из возможных реализаций:

```
class Pair<T, U> {
    readonly first: T;
    readonly second: U;
```

```
    constructor(first: T, second: U) {
        this.first = first;
        this.second = second;
    }
}
```

9.3. Обход произвольной структуры данных

1. Эта реализация очень похожа на реализацию центрированного обхода; мы просто выдаем `root.value`, прежде чем выдавать левое поддерево:

```
function* preOrderIterator<T>(root: BinaryTreeNode<T>):
    IterableIterator<T> {
    yield root.value;

    if (root.left) {
        for (const value of preOrderIterator(root.left)) {
            yield value;
        }
    }

    if (root.right) {
        for (const value of preOrderIterator(root.right)) {
            yield value;
        }
    }
}
```

2. В данной реализации для обхода массива в обратную сторону используется цикл `for`, так что вызывающей стороне этого делать не нужно:

```
function* backwardsArrayIterator<T>(array: T[]): IterableIterator<T> {
    for (let i = array.length - 1; i >= 0; i--) {
        yield array[i];
    }
}
```

9.4. Потоковая обработка данных

1. Одна из возможных реализаций:

```
function* drop<T>(iter: Iterable<T>, n: number):
    IterableIterator<T> {
    for (const value of iter) {
        if (n-- > 0) continue;

        yield value;
    }
}
```

2. Можно описать генератор `count()` — счетчик, выдающий числа, начиная с 1 и до бесконечности. А затем, применив `drop()`, отбросить первые пять значений из выдаваемого им потока данных, после чего с помощью `take()` отобрать следующие пять:

```
function* count(): IterableIterator<number> {
    let n: number = 0;

    while (true) {
        n++;
        yield n;
    }
}

for (let value of take(drop(count(), 5), 5)) {
    console.log(value);
```

10 Обобщенные алгоритмы и итераторы

В этой главе

- Использование операций `map()`, `filter()` и `reduce()` не только для массивов.
- Решение широкого спектра задач с помощью набора распространенных алгоритмов.
- Обеспечение поддержки обобщенным типом данных нужного контракта.
- Реализация разнообразных алгоритмов с помощью различных категорий итераторов.
- Реализация адаптивных алгоритмов.

Эта глава всецело посвящена обобщенным алгоритмам, пригодным для повторного использования, подходящим для разнообразных типов и структур данных.

Мы рассмотрели по одной версии каждой из операций `map()`, `filter()` и `reduce()` в главе 5, когда обсуждали функции высшего порядка. Эти функции работают с массивами, но, как мы видели в предыдущих главах, итераторы — прекрасная абстракция для работы с любой структурой данных. Мы начнем с реализации обобщенных версий трех вышеупомянутых алгоритмов, работающих с итераторами, а значит, применимых для обработки бинарных деревьев, списков, массивов и любых других итерируемых структур данных.

Операции `map()`, `filter()` и `reduce()` не единственные в своем роде. Мы поговорим и о других обобщенных алгоритмах и библиотеках алгоритмов, доступных в большинстве современных языков программирования. Мы увидим, почему лучше заменить большинство циклов вызовами библиотечных алгоритмов. Мы также немного поговорим о текучих API и удобных для пользователя интерфейсах алгоритмов.

Далее мы пройдемся по ограничениям типов-параметров; обобщенные структуры данных и алгоритмы могут задавать возможности, которые должны присутствовать в их типах-параметрах. Подобная специализация приводит к несколько менее универсальным структурам данных и алгоритмам, пригодным к использованию уже не везде.

Мы подробнее обсудим итераторы и поговорим об их различных категориях. Чем у́же специализирован итератор, тем более эффективные алгоритмы возможны с его участием. Впрочем, не все структуры данных способны поддерживать специализированные итераторы.

Наконец, мы коротко пробежимся по адаптивным алгоритмам. Они позволяют создавать более универсальные, но менее эффективные реализации для итераторов, обладающих меньшим количеством возможностей, и более эффективные, но менее универсальные реализации для итераторов с бо́льшим количеством возможностей.

10.1. Улучшенные операции map(), filter() и reduce()

В главе 5 мы говорили об операциях `map()`, `filter()` и `reduce()` и рассмотрели одну из возможных реализаций каждой из них. Эти алгоритмы представляют собой функции высшего порядка, поскольку принимают в качестве аргумента другую функцию, применяя ее к последовательности данных.

Операция `map()` применяет функцию к каждому элементу последовательности и возвращает результаты. Операция `filter()` применяет функцию фильтрации к каждому элементу и возвращает только те из них, для которых эта функция вернула `true`. Операция `reduce()` группирует все значения в последовательности с помощью функции и возвращает в виде результата одно значение.

В нашей реализации из главы 5 использовался обобщенный тип-параметр `T`, а последовательности были представлены в виде массивов элементов типа `T`.

10.1.1. Операция map()

Вспомним, как мы реализовали операцию `map()`. У нас было два типа-параметра: `T` и `U`. Функция принимает как аргумент массив значений типа `T` и функцию, переводящую из `T` в `U` в качестве второго аргумента. Она возвращает массив значений типа `U`, как показано в листинге 10.1.

Теперь, воспользовавшись нашими знаниями об итераторах и генераторах, посмотрим в листинге 10.2, как реализовать `map()` так, чтобы она могла работать с любым `Iterable<T>`, а не только с массивами.

В то время как область действия первоначальной реализации ограничивалась массивами, эта может работать с любой структурой данных, предоставляющей итератор. Кроме того, код стал намного компактнее.

Листинг 10.1. Операция map()

```
function map<T, U>(items: T[], func: (item: T) => U): U[] {
    let result: U[] = [];

    for (const item of items) {
        result.push(func(item));
    }

    return result;
}
```

Операция map() принимает на входе массив элементов типа T и функцию, переводящую из T в U, и возвращает массив значений типа U

В начале массив значений типа U пуст

Для каждого результата вставляем результат выполнения func(item) в массив значений типа U

Возвращаем массив значений типа U

Листинг 10.2. Операция map() с итератором

```
function* map<T, U>(iter: Iterable<T>, func: (item: T) => U):
    IterableIterator<U> {
    for (const value of iter) {
        yield func(value);
    }
}
```

Функция map() теперь представляет собой генератор, принимающий в качестве первого аргумента Iterable<T>

Функция map() возвращает IterableIterator<U>

Заданная функция применяется к каждому извлеченному из итератора значению, и результат выдается наружу

10.1.2. Операция filter()

Проделаем то же самое с `filter()` (листинг 10.3). Наша исходная реализация ожидала на входе массив элементов типа `T` и предикат. Напомню, что *предикат* — это функция, принимающая один элемент какого-то типа и возвращающая `boolean`. Если данная функция возвращает `true` для переданного ей значения, то говорят, что значение удовлетворяет предикату.

Листинг 10.3. Операция filter()

```
function filter<T>(items: T[], pred: (item: T) => boolean): T[] {
    let result: T[] = [];

    for (const item of items) {
        if (pred(item)) {
            result.push(item);
        }
    }

    return result;
}
```

Функция filter() принимает в качестве аргументов массив значений типа T и предикат (функцию, переводящую из T в boolean)

Если предикат возвращает true, то добавляем элемент в итоговый массив, в противном случае пропускаем его

Как и в случае с операцией `map()`, мы воспользуемся `Iterable<T>` вместо массива и реализуем этот `Iterable` в виде генератора, выдающего значения, удовлетворяющие предикату, как показано в листинге 10.4.

Листинг 10.4. Операция filter() с итератором

```
function* filter<T>(iter: Iterable<T>, pred: (item: T) => boolean):
    IterableIterator<T> {
    for (const value of iter) {
        if (pred(value)) {
            yield value;
        }
    }
}
```

Функция filter() теперь представляет собой генератор, принимающий в качестве первого аргумента Iterable<T>

Функция filter() возвращает IterableIterator<U>

Если значение удовлетворяет предикату, то выдается наружу

Опять получилась более лаконичная реализация, способная работать не только с массивами. И наконец, модифицируем функцию `reduce()`.

10.1.3. Операция reduce()

Наша первоначальная реализация `reduce()` ожидала на входе массив элементов типа `T`, начальное значение типа `T` (на случай, если массив окажется пуст) и операцию `op()`. Эта операция представляет собой функцию, которая принимает в качестве аргументов два значения типа `T` и возвращает одно значение типа `T`. Операция `reduce()` применяет операцию к начальному значению и первому элементу массива, сохраняет результат, применяет ту же операцию к результату и следующему элементу массива и т. д. (листинг 10.5)

Листинг 10.5. Операция reduce()

```
function reduce<T>(items: T[], init: T, op: (x: T, y: T) => T): T {
    let result: T = init;

    for (const item of items) {
        result = op(result, item);
    }

    return result;
}
```

Функция reduce() принимает в качестве аргументов массив значений типа T, начальное значение и операцию группировки двух значений типа T в одно

Все элементы массива группируются с промежуточным итогом с помощью заданной операции

Эту функцию можно переписать так, чтобы вместо массива использовался `Iterable<T>` и она могла работать с любой последовательностью, как показано в листинге 10.6. В данном случае генератор нам не нужен. В отличие от двух предыдущих функций, `reduce()` возвращает не последовательность элементов, а одно значение.

Листинг 10.6. Операция reduce() с итератором

```
function reduce<T>(iter: Iterable<T>, init: T,
    op: (x: T, y: T) => T): T {
    let result: T = init;

    for (const value of iter) {
        result = op(result, value);
    }

    return result;
}
```

Вместо массива значений типа T reduce() принимает в качестве первого аргумента Iterable<T>

Остальная часть реализации не поменялась.

10.1.4. Конвейер filter()/reduce()

Посмотрим, как объединить эти алгоритмы в конвейер, выбирающий из бинарного дерева только четные числа и суммирующий их. Воспользуемся классом `BinaryTreeNode<T>` из главы 9 с его центрированным обходом дерева и сцепим его с фильтром четных чисел и функцией `reduce()`, в которой в качестве операции применяется сложение (листинг 10.7).

Листинг 10.7. Конвейер filter()/reduce()

```
let root: BinaryTreeNode<number> = new BinaryTreeNode(1);
root.left = new BinaryTreeNode(2);
root.left.right = new BinaryTreeNode(3);
root.right = new BinaryTreeNode(4);

const result: number =
    reduce(
        filter(
            inOrderIterator(root),
            (value) => value % 2 == 0),
        0, (x, y) => x + y);

console.log(result);
```

Тот же пример бинарного дерева, что и в главе 9

Получаем IterableIterator<number> для обхода дерева по порядку

Фильтруем с помощью лямбда-выражения, возвращающего true для четных чисел

Выполняем свертку, начиная с начального значения 0, с помощью лямбда-выражения, суммирующего два числа

Этот пример — живое подтверждение эффективности обобщенных типов. Вместо того чтобы реализовывать новую функцию для обхода бинарного дерева и суммирования четных чисел, мы просто формируем конвейер обработки специально для нужного сценария.

10.1.5. Упражнения

1. Создайте конвейер для обработки итерируемого объекта типа `string`: конкатенации всех непустых строк.
2. Создайте конвейер для обработки итерируемого объекта типа `number`: выбора всех нечетных чисел и возведения их в квадрат.

10.2. Распространенные алгоритмы

Мы обсудили алгоритмы `map()`, `filter()` и `reduce()`, а также был упомянут `take()` в главе 9. В конвейерах часто используются и многие другие алгоритмы. Перечислю некоторые из них. Мы не станем изучать реализации — я просто опишу, какие аргументы (помимо `Iterable`) они получают и как обрабатывают данные. Вдобавок я перечислю различные названия, под которыми эти алгоритмы могут встречаться:

- `map()` принимает на входе последовательность значений типа `T` и функцию `(value: T) => U`, а возвращает последовательность значений типа `U` после применения ко всем значениям входной последовательности этой функции. Она также встречается под названиями `fmap()`, `select()`;
- `filter()` принимает на входе последовательность значений типа `T` и предикат `(value: T) => boolean`, а возвращает последовательность значений типа `T`, включающую все элементы, для которых этот предикат возвращает `true`. Встречается также под названием `where()`;
- `reduce()` принимает на входе последовательность значений типа `T` и операцию группировки двух значений типа `T` в одно `(x: T, y: T) => T`. После группировки всех элементов последовательности с помощью этой операции `reduce()` возвращает одно значение типа `T`. Она также встречается под названиями `fold()`, `collect()`, `accumulate()`, `aggregate()`;
- `any()` принимает на входе последовательность значений типа `T` и предикат `(value: T) => boolean`. Она возвращает `true`, если хотя бы один элемент последовательности удовлетворяет предикату;
- `all()` принимает на входе последовательность значений типа `T` и предикат `(value: T) => boolean`. Она возвращает `true`, если все элементы последовательности удовлетворяют предикату;
- `none()` принимает на входе последовательность значений типа `T` и предикат `(value: T) => boolean`. Она возвращает `true`, если ни один из элементов последовательности не удовлетворяет предикату;
- `take()` принимает на входе последовательность значений типа `T` и число `n`. Она возвращает последовательность, состоящую из первых `n` элементов исходной последовательности. Встречается также под названием `limit()`;
- `drop()` принимает на входе последовательность значений типа `T` и число `n`. Она возвращает последовательность, состоящую из всех элементов исходной последовательности, за исключением первых `n`. Встречается также под названием `skip()`;
- `zip()` принимает на входе последовательность значений типа `T` и последовательность значений типа `U`, а возвращает последовательность, состоящую из пар значений `T` и `U`, по сути, склеивая две входные последовательности.

Существует множество других алгоритмов для сортировки, обращения, разбиения и конкатенации последовательностей. К счастью, эти алгоритмы настолько полезны и применимы в таком количестве областей, что реализовывать их самостоя-

тельно не требуется. Для большинства языков программирования существуют библиотеки с готовыми реализациями. В JavaScript есть пакеты `underscore.js` и `lodash`, в каждом из которых множество подобных алгоритмов. (На момент написания данной книги эти библиотеки не поддерживали итераторы — только встроенные типы массивов и объектов JavaScript.) В Java их можно найти в пакете `java.util.stream`. В C# они располагаются в пространстве имен `System.Linq`. В C++ — в заголовочном файле стандартной библиотеки `<algorithm>`.

10.2.1. Алгоритмы вместо циклов

Возможно, вы удивитесь, но есть хорошее эмпирическое правило: всякий раз, когда пишете алгоритм, проверьте, не существует ли библиотечного алгоритма или конвейера для этой задачи. Обычно циклы пишутся для обработки последовательностей — именно для того, для чего служат обсуждавшиеся выше алгоритмы.

Библиотечные алгоритмы предпочтительнее пользовательских циклов потому, что вероятность ошибки меньше. Библиотечные алгоритмы — проверенные и реализованы эффективным образом, и их применение позволяет получить более понятный код благодаря явному указанию операций.

В этой книге мы рассмотрели несколько реализаций, чтобы лучше понять внутренние механизмы, но реализовывать алгоритмы самостоятельно приходится редко. Если вам встретится задача, которая не под силу существующим алгоритмам, то лучше создать обобщенную, повторно используемую реализацию, чем одноразовую специализированную.

10.2.2. Реализация текучего конвейера

Большинство библиотек предоставляют текучий API для объединения алгоритмов в конвейер. Текучие API — это API, в основе которых лежит сцепление, значительно упрощающее чтение кода. Посмотрим на разницу между текучим и нетекучим API на примере конвейера фильтрации/свертки из раздела 10.1.4 (листинг 10.8).

Листинг 10.8. Конвейер filter/reduce

```
let root: BinaryTreeNode<number> = new BinaryTreeNode(1);
root.left = new BinaryTreeNode(2);
root.left.right = new BinaryTreeNode(3);
root.right = new BinaryTreeNode(4);

const result: number =
    reduce(
        filter(
            inOrderBinaryTreeIterator(root),
            (value) => value % 2 == 0),
        0, (x, y) => x + y);

console.log(result);
```

И хоть мы сначала применяем операцию `filter()` и затем передаем результат в операцию `reduce()`, при чтении кода слева направо увидим `reduce()` перед `filter()`. Кроме того, довольно непросто разобраться, какие аргументы относятся к той или иной функции в конвейере. Текучий API намного облегчает чтение кода.

В настоящее время все наши алгоритмы принимают как первый аргумент объект итерируемого типа и возвращают итератор. Объектно-ориентированное программирование позволит усовершенствовать наш API. Можно собрать все наши алгоритмы в класс-обертку для итерируемого объекта. А затем вызывать любой из итерируемых объектов без явного указания его в качестве первого аргумента, ведь теперь итерируемый объект является членом класса. Проделаем это для `map()`, `filter()` и `reduce()`, сгруппировав их в новый класс `FluentIterable<T>`, служащий оберткой для объекта `Iterable`, как показано в листинге 10.9.

Листинг 10.9. Текучий Iterable

```
class FluentIterable<T> {
    iter: Iterable<T>;

    constructor(iter: Iterable<T>) {
        this.iter = iter;
    }

    *map<U>(func: (item: T) => U): IterableIterator<U> {
        for (const value of this.iter) {
            yield func(value);
        }
    }

    *filter(pred: (item: T) => boolean): IterableIterator<T> {
        for (const value of this.iter) {
            if (pred(value)) {
                yield value;
            }
        }
    }

    reduce(init: T, op: (x: T, y: T) => T): T {
        let result: T = init;

        for (const value of this.iter) {
            result = op(result, value);
        }

        return result;
    }
}
```

Класс FluentIterable<T> служит оберткой для Iterable<T>

Реализации map(), filter() и reduce() аналогичны предыдущим, но вместо итерируемого объекта в качестве первого аргумента в них используется итерируемый this.iter

На основе `Iterable<T>` можно создать `FluentIterable<T>`, поэтому мы можем переписать наш конвейер `filter/reduce` в более текучем виде. Создадим объект `FluentIterable<T>`, вызовем для него `filter()`, создадим на основе результатов его работы новый объект `FluentIterable<T>` и вызовем для него `reduce()`, как показано в листинге 10.10.

Листинг 10.10. Текучий конвейер filter/reduce

```
let root: BinaryTreeNode<number> = new BinaryTreeNode(1);
root.left = new BinaryTreeNode(2);
root.left.right = new BinaryTreeNode(3);
root.right = new BinaryTreeNode(4);

const result: number =
    new FluentIterable(
        new FluentIterable(
            inOrderIterator(root)
        ).filter((value) => value % 2 == 0)
    ).reduce(0, (x, y) => x + y);

console.log(result);
```

Получаем объект для итерации по бинарному дереву из inOrderIterator и инициализируем им FluentIterable

Вызываем для FluentIterable<T> функцию filter(), после чего создаем на основе возвращаемых ей результатов другой FluentIterable<T>

Наконец, вызываем для FluentIterable<T> функцию reduce() и получаем итоговый результат

Теперь `filter()` встречается перед `reduce()`, и совершенно ясно, что аргументы относятся к этой функции. Единственная проблема состоит в необходимости создавать новый объект `FluentIterable<T>` после каждого вызова функции. Можно усовершенствовать данный API, если переписать функции `map()` и `filter()` так, чтобы они возвращали `FluentIterable<T>` вместо `IterableIterator<T>`. Обратите внимание: менять метод `reduce()` не требуется, поскольку `reduce()` возвращает единственное значение типа `T`, а не итерируемый объект.

Но поскольку мы используем генераторы, то просто поменять возвращаемый тип нельзя. Смысл существования генераторов состоит в том, чтобы обеспечивать удобный синтаксис для функций, но они всегда возвращают `IterableIterator<T>`. Вместо этого мы можем перенести реализации в два приватных метода: `mapImpl()` и `filterImpl()` — и производить преобразование из `IterableIterator<T>` в `FluentIterable<T>` в публичных методах `map()` и `filter()`, как показано в листинге 10.11.

Листинг 10.11. Усовершенствованный класс FluentIterable

```
class FluentIterable<T> {
    iter: Iterable<T>;

    constructor(iter: Iterable<T>) {
        this.iter = iter;
    }

    map<U>(func: (item: T) => U): FluentIterable<U> {
        return new FluentIterable(this.mapImpl(func));
    }

    private *mapImpl<U>(func: (item: T) => U): IterableIterator<U> {
        for (const value of this.iter) {
            yield func(value);
        }
    }
```

Метод map() передает полученный аргумент методу mapImpl() и преобразует результат во FluentIterable

Метод mapImpl() — это исходная реализация map() в виде генератора

```
    filter<U>(pred: (item: T) => boolean): FluentIterable<T> {
        return new FluentIterable(this.filterImpl(pred));
    }

    private *filterImpl(pred: (item: T) => boolean): IterableIterator<T> {
        for (const value of this.iter) {
            if (pred(value)) {
                yield value;
            }
        }
    }

    reduce(init: T, op: (x: T, y: T) => T): T {
        let result: T = init;

        for (const value of this.iter) {
            result = op(result, value);
        }

        return result;
    }
}
```

Подобно map(), filter() передает полученный аргумент методу filterImpl() и преобразует результат во FluentIterable

Метод filterImpl() — это исходная реализация filter() в виде генератора

Метод reduce() не меняется, поскольку не возвращает итератор

Благодаря этой усовершенствованной реализации становится удобнее сцеплять алгоритмы, поскольку они все теперь возвращают `FluentIterable`, в котором все алгоритмы описаны в виде методов, как показано в листинге 10.12.

Листинг 10.12. Усовершенствованный текучий конвейер filter/reduce

```
let root: BinaryTreeNode<number> = new BinaryTreeNode(1);
root.left = new BinaryTreeNode(2);
root.left.right = new BinaryTreeNode(3);
root.right = new BinaryTreeNode(4);

const result: number =
    new FluentIterable(inOrderIterator(root))
    .filter((value) => value % 2 == 0)
    .reduce(0, (x, y) => x + y);

console.log(result);
```

Достаточно создать явным образом новый FluentIterable на основе исходного итератора для бинарного дерева только один раз

filter() — метод класса FluentIterable и сам возвращает FluentIterable

Можно вызвать метод reduce() возвращенного методом filter() результата

Теперь в этом по-настоящему текучем варианте код легко читается слева направо, и синтаксис сцепления произвольного количества составляющих конвейер алгоритмов выглядит вполне естественно. Подобный подход применяется в большинстве библиотек алгоритмов, максимально упрощая сцепление нескольких алгоритмов.

В зависимости от языка программирования один из недостатков текучих API — загромождение класса `FluentIterable` методами для всех алгоритмов, что усложняет его расширение. Если он включен в библиотеку, то у вызывающего кода нет возможности добавить новый алгоритм без модификации класса. В C# существуют так называемые методы расширения (extension methods), которые позволяют добавлять в класс или интерфейс методы, не прибегая к модификации его кода. Впрочем, не во

всех языках присутствуют подобные возможности. Тем не менее в большинстве случаев лучше использовать уже существующую библиотеку алгоритмов, а не реализовывать новую с нуля.

10.2.3. Упражнения

1. Расширьте класс `FluentIterable`, добавив в него алгоритм `take()`, возвращающий первые `n` элементов из итератора.
2. Расширьте класс `FluentIterable`, добавив в него алгоритм `drop()`, пропускающий первые `n` элементов из итератора и возвращающий все остальные.

10.3. Ограничение типов-параметров

Мы уже видели, как обобщенные структуры данных задают форму данных, не зависящую от конкретного типа-параметра `T`. Мы также рассмотрели набор алгоритмов, использующих итераторы для обработки последовательностей значений типа `T`, вне зависимости от того, какой конкретно это тип. Теперь в листинге 10.13 рассмотрим сценарий, при котором конкретный тип данных важен: обобщенную функцию `renderAll()`, принимающую в качестве аргумента `Iterable<T>` и вызывающую метод `render()` для каждого возвращаемого итератором элемента.

Листинг 10.13. Набросок renderAll()

```
function renderAll<T>(iter: Iterable<T>): void {
    for (const item of iter) {
        item.render();
    }
}
```

Функция renderAll() принимает в качестве аргумента Iterable<T>

Для каждого возвращаемого итератором элемента вызывается метод render()

Попытка компиляции этой функции завершается следующим сообщением об ошибке: `Property 'render' does not exist on type 'T'` (в типе `'T'` отсутствует свойство `'render'`).

Мы пытаемся вызвать метод `render()` обобщенного типа `T`, в котором такого метода может и не быть. При подобном сценарии необходимо *ограничить* тип `T` исключительно конкретными воплощениями, содержащими метод `render()`.

ОГРАНИЧЕНИЯ ТИПОВ-ПАРАМЕТРОВ

> Ограничения сообщают компилятору о возможностях, которые обязательно должны быть у типа-аргумента. Без ограничений тип-аргумент может быть любым. Когда требуется наличие у обобщенного типа данных определенных членов класса, именно с помощью ограничений мы сокращаем множество допустимых типов до тех, в которых есть эти требуемые члены.

В нашем случае можно описать интерфейс `IRenderable`, в котором объявлен метод `render()`, как показано в листинге 10.14. Далее можно добавить ограничение

на тип `T` с помощью ключевого слова `extends`, указав тем самым компилятору, что допустимы только типы-аргументы, воплощающие `IRenderable`.

Листинг 10.14. Функция renderAll с ограничениями

```
interface IRenderable {
    render(): void;
}

function renderAll<T extends IRenderable>(iter: Iterable<T>): void {
    for (const item of iter) {
        item.render();
    }
}
```

Воплощающие интерфейс IRenderable типы должны включать метод render()

T extends IRenderable дает компилятору указание принимать только такие конкретные воплощения типа T, которые реализуют интерфейс IRenderable

10.3.1. Обобщенные структуры данных с ограничениями типа

Для большинства обобщенных структур данных не требуется ограничение типов-параметров. В связном списке, дереве и массиве можно хранить значения любого типа. Однако существует несколько исключений, в частности хешированное множество.

Структура данных множества моделирует множество в математическом смысле, поэтому в нем хранятся уникальные значения, а дубликаты отбрасываются. Структуры данных множества обычно включают методы для объединения, пересечения и вычитания из других множеств, а также позволяют проверять, включено ли заданное значение в множество. Чтобы выполнить подобную проверку, можно сравнить это значение с каждым из элементов множества, хоть это не слишком эффективный подход. В худшем случае сравнение с каждым из элементов множества требует обхода всего множества. Подобный обход выполняется за *линейное время*, $O(n)$. Освежить знания об этих обозначениях вы можете во врезке «Нотация О большое» далее.

Наиболее эффективная реализация — хеширование всех значений и хранение их в структуре данных «ключ — значение», например в хеш-карте или словаре. Подобные структуры более эффективны, они могут извлекать значения за *константное время*, $O(1)$. Хешированное множество служит оберткой для хеш-карты, обеспечивая эффективную проверку на включение элемента. Но у него есть ограничение: тип `T` должен предоставлять хеш-функцию, принимающую значение типа `T` и возвращающую его хеш-значение типа `number`.

В некоторых языках программирования хеширование всех значений осуществимо за счет описания метода хеширования в высшем типе. Например, высший тип Java `Object` включает метод `hashCode()`, а высший тип C# `Object` включает метод `GetHashCode()`. Но если язык не предоставляет подобной возможности, то необходимо ограничение типа, чтобы в наших структурах данных могли храниться только типы, допускающие хеширование. Например, можно описать интерфейс `IHashable` и воспользоваться им в качестве ограничения типа для типа ключей наших обобщенных хеш-карты или словаря.

Нотация О большое

С помощью нотации О большое указываются оценки сверху времени и памяти, необходимые для работы функции, при стремлении аргументов этой функции к конкретному значению n. Мы не будем слишком углубляться в данный вопрос, а просто рассмотрим несколько распространенных видов оценок сверху и выясним, что они означают.

Константное время (constant time), $O(1)$, означает, что время выполнения функции не зависит от количества обрабатываемых элементов. Например, функция `first()`, извлекающая из последовательности первый элемент, работает одинаково быстро для последовательностей из 2 и 2 000 000 элементов.

Логарифмическое время (logarithmic time), $O(\log n)$, означает, что время выполнения функции пропорционально логарифму[1] от количества обрабатываемых элементов, что очень эффективно даже для больших значений n. Примером таких алгоритмов может служить двоичный поиск в отсортированной последовательности.

Линейное время (linear time), $O(n)$, означает, что время выполнения функции растет пропорционально объему входных данных. За линейное время выполняется проход в цикле по последовательности, например, для определения, все ли элементы последовательности удовлетворяют некоему предикату.

Квадратичное время (quadratic time), $O(n^2)$, означает, что функция намного менее эффективна, чем при линейном времени, ведь время выполнения растет гораздо быстрее размера входных данных. За квадратичное время выполняются два вложенных прохода в цикле по последовательности.

Линейно-логарифмическое время (linearithmic time), $O(n \log n)$, не столь эффективно, как линейное, но эффективнее квадратичного. Сложность наиболее эффективных алгоритмов сортировки сравнением составляет $O(n \log n)$; отсортировать последовательность в одном цикле нельзя, но можно сделать это быстрее, чем с помощью двух вложенных циклов.

Подобно тому как сложность по времени задает оценку сверху для времени выполнения функции в зависимости от размера входных данных, так и пространственная сложность алгоритма задает оценку сверху для объема дополнительной памяти, которая нужна функции при росте входных данных.

Константный объем пространства (constant space), $O(1)$, означает, что функции не требуется больше памяти при росте объема входных данных. Например, нашей функции `max()` нужно немного дополнительного места для хранения промежуточного максимума и итератора, но количество места не зависит от размера последовательности.

Линейный объем пространства (linear space), $O(n)$, означает, что требуемое функции количество памяти пропорционально размеру входных данных. В качестве примера подобной функции можно привести нашу исходную функцию обхода бинарного дерева `inOrder()`, копирующую значения всех узлов в массив для итерации по дереву.

[1] Обычно по основанию 2. Другое основание соответствует просто умножению на фиксированный множитель. — *Примеч. пер.*

10.3.2. Обобщенные алгоритмы с ограничениями типа

У алгоритмов обычно больше ограничений на типы, чем у структур данных. Отсортировать набор значений можно при наличии способа сравнить их. Аналогично, чтобы определить минимальный или максимальный элемент последовательности, ее элементы должны быть доступны для сравнения.

Рассмотрим одну из возможных реализаций обобщенного алгоритма `max()` в листинге 10.15. Для начала объявим интерфейс `IComparable<T>`, ограничив им алгоритм. В этом интерфейсе объявлен один метод — `compareTo()`.

Листинг 10.15. Интерфейс IComparable

```
enum ComparisonResult {
    LessThan,
    Equal,
    GreaterThan
}

interface IComparable<T> {
    compareTo(value: T): ComparisonResult;
}
```

Перечисляемый тип данных отражает результат сравнения

В интерфейсе IComparable объявлен метод compareTo(), сравнивающий текущий экземпляр с другим значением того же типа и возвращающий ComparisonResult

Теперь реализуем обобщенный алгоритм `max()`, который получает на входе итератор для прохода по набору значений `IComparable` и возвращает максимальный элемент, как показано в листинге 10.16. Мы должны учесть случай, когда итератор не содержит никаких значений и `max()` должен вернуть `undefined`. Поэтому вместо того, чтобы использовать цикл `for...of`, мы будем передвигать итератор вручную, с помощью `next()`.

Листинг 10.16. Алгоритм max()

```
function max<T extends IComparable<T>>(iter: Iterable<T>)
    : T | undefined {
    let iterator: Iterator<T> = iter[Symbol.iterator]();

    let current: IteratorResult<T> = iterator.next();

    if (current.done) return undefined;

    let result: T = current.value;

    while (true) {
        current = iterator.next();

        if (current.done) return result;

        if (current.value.compareTo(result) ==
            ComparisonResult.GreaterThan) {
            result = current.value;
        }
    }
}
```

Вызываем next() один раз для получения первого значения

Алгоритм max() ограничивает возможный тип T типами, реализующими интерфейс IComparable<T>

Получаем Iterator<T> из аргумента Iterable<T>

Если первого значения нет, то возвращаем undefined

Задаем начальное значение переменной result — первое значение, возвращенное итератором

Когда итератор завершается, возвращаем result

Всякий раз, когда текущее значение больше, чем нынешний максимум, присваиваем переменной result текущее значение

Многие алгоритмы, такие как `max()`, выдвигают определенные требования к типам, с которыми работают. Есть и другая возможность — сделать операцию сравнения аргументом самой функции в противовес ограничению обобщенного типа. Вместо `IComparable<T>` функция `max()` может получать второй аргумент — функцию `compare()`, переводящую два аргумента типа `T` в `ComparisonResult`, как показано в листинге 10.17.

Листинг 10.17. Алгоритм max() с аргументом compare()

```
function max<T>(iter: Iterable<T>,
    compare: (x: T, y: T) => ComparisonResult)
    : T | undefined {
    let iterator: Iterator<T> = iter[Symbol.iterator]();

    let current: IteratorResult<T> = iterator.next();

    if (current.done) return undefined;

    let result: T = current.value;

    while (true) {
        current = iterator.next();

        if (current.done) return result;

        if (compare(current.value, result)
            == ComparisonResult.GreaterThan) {
            result = current.value;
        }
    }
}
```

Функция compare() принимает два аргумента типа T и возвращает ComparisonResult

Вместо метода IComparable.compareTo() вызывается переданный аргумент compare()

Преимущество этой реализации заключается в отсутствии ограничений на тип `T`, вследствие чего можно передать сюда любую функцию сравнения. Недостатком является то, что для типов, обладающих естественным порядком (чисел, температур, расстояний и т. д.), приходится указывать функцию сравнения явным образом. В хороших библиотеках алгоритмов обычно есть обе версии алгоритма: одна, в которой используется естественное сравнение для данного типа, и вторая, в которую вызывающая сторона может передать свою собственную функцию сравнения.

Чем больше алгоритму известно о методах и свойствах обрабатываемого типа `T`, тем полнее можно использовать их в его реализации. Далее посмотрим на применение итераторов для повышения эффективности реализаций алгоритмов.

10.3.3. Упражнение

Реализуйте обобщенную функцию `clamp()`, принимающую на входе значение, а также аргументы `low` и `high`. Если значение входит в диапазон от `low` до `high`, то функция возвращает значение. Если оно меньше `low`, то возвращает `low`. Если же превышает `high`, то возвращает `high`. Воспользуйтесь описанным в этом разделе интерфейсом `IComparable`.

10.4. Эффективная реализация reverse и других алгоритмов с помощью итераторов

До сих пор мы обсуждали алгоритмы линейной обработки последовательностей. Так, `map()`, `filter()`, `reduce()` и `max()` проходят в цикле по последовательности значений от начала до конца. Все они выполняются за линейное время (пропорциональное размеру последовательности) и с константным объемом пространства. (Требования к памяти не меняются независимо от размера последовательности.) Рассмотрим еще один алгоритм — `reverse()`.

Этот алгоритм «переворачивает» входную последовательность: последний элемент становится первым, предпоследний — вторым и т. д. Один из вариантов реализации `reverse()` — поместить все входные элементы в стек, а затем извлечь их в обратном порядке, как показано на рис. 10.1 и в листинге 10.18.

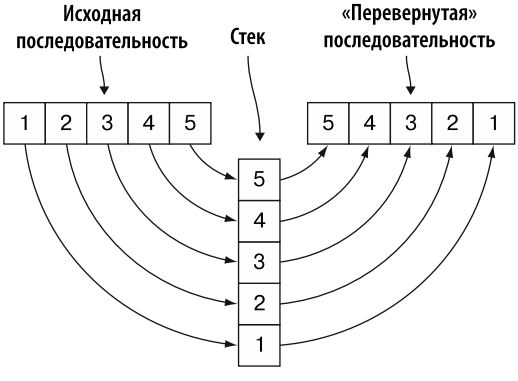


Рис. 10.1. «Переворачивание» последовательности с помощью стека: элементы исходной последовательности помещаются в стек, а затем извлекаются в обратном порядке

Листинг 10.18. Реализация reverse() с помощью стека

```
function *reverse<T>(iter: Iterable<T>): IterableIterator<T> {
    let stack: T[] = [];

    for (const value of iter) {
        stack.push(value);
    }

    while (true) {
        let value: T | undefined = stack.pop();

        if (value == undefined) return;

        yield value;
    }
}
```

Функция reverse() представляет собой генератор, следующий тому же паттерну, что и остальные наши алгоритмы

У массивов JavaScript всегда есть методы push() и pop(), поэтому воспользуемся массивом в качестве стека

Помещаем все значения из последовательности в стек

Извлекаем значение из стека. Если стек пуст, то оно равно undefined

Когда стек оказывается пуст, выполняем оператор return — работа функции завершена

Выдаем значение с помощью оператора yield и переходим к следующей итерации цикла

Это простая, но не самая эффективная реализация. Она выполняется за линейное время, однако требует и линейного объема памяти. Чем больше входная последовательность, тем больше памяти понадобится данному алгоритму на вставку элементов в стек.

Пока ненадолго забудем об итераторах и попробуем эффективно реализовать обращение массива, как показано в листинге 10.19. Его можно произвести, не прибегая к созданию дополнительных структур данных, таких как стек, просто меняя элементы массива местами с двух концов одновременно (рис. 10.2).

Как видите, эта реализация эффективнее предыдущей. Зависимость от времени по-прежнему линейная, поскольку приходится обращаться к каждому из элементов последовательности (без чего перевернуть последовательность не получится), но для работы ей требуется константный объем пространства. В отличие от предыдущей версии, где требовался стек такого же размера, как и входная последовательность, в этой достаточно одной переменной `temp` типа `T`, вне зависимости от размера входных данных.

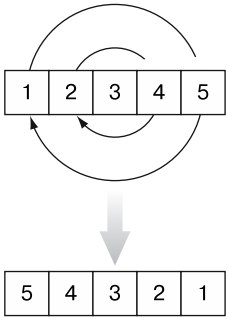


Рис. 10.2. Обращение массива путем обмена его элементов местами без создания дополнительных структур данных

Можно ли обобщить этот пример и создать эффективный алгоритм обращения произвольной структуры данных? Можно, однако нам придется внести небольшие поправки в наше представление об итераторах. TypeScript предоставляет интерфейсы `Iterator<T>`, `Iterable<T>` и их сочетание `IterableIterator<T>` в рамках стандарта ES6 JavaScript. Мы же сейчас выйдем за его пределы и рассмотрим некоторые не включенные в него итераторы.

Листинг 10.19. Реализация reverse() для массива

```
function reverse<T>(values: T[]): void {
    let begin: number = 0;
    let end: number = values.length;

    while (begin < end) {
        const temp: T = values[begin];
        values[begin] = values[end - 1];
        values[end - 1] = temp;

        begin++;
        end--;
    }
}
```

Эта версия reverse() ожидает на входе массив элементов типа T, а не итерируемый объект

Изначально переменные begin и end указывают на начало и конец массива

Повторяем, пока они (begin и end) не совпадут или не пройдут друг друга

Меняем местами значение в begin со значением в end – 1 (изначально end указывал на один элемент дальше последнего элемента массива)

Увеличиваем индекс begin на единицу и уменьшаем индекс end на единицу

10.4.1. Стандартные блоки, из которых состоят итераторы

Итераторы JavaScript позволяют извлечь значение и перейти к следующему, повторяя эти действия до тех пор, пока не закончится последовательность. Для работы же алгоритма без создания дополнительных структур данных нужны дополнительные возможности. Нужно уметь не только прочитать значение на заданной позиции,

но и установить его. В случае нашего алгоритма `reverse()` мы начинаем с обоих концов последовательности и заканчиваем работу в ее середине, поэтому итератор сам по себе не сможет определить, когда обработка будет завершена. Нам известно, что выполнение `reverse()` закончено, когда `begin` и `end` минуют друг друга, следовательно, нужен способ сравнения двух итераторов.

В целях создания эффективных алгоритмов переопределим наши итераторы в виде набора интерфейсов, каждый из которых описывает дополнительные возможности. Сначала опишем интерфейс `IReadable<T>`, включающий публичный метод `get()`, который возвращает значение типа `T`. Мы будем использовать этот метод для чтения значения из итератора. Кроме того, опишем интерфейс `IIncrementable<T>`, включающий публичный метод `increment()`, подходящий для продвижения итератора вперед, как показывает листинг 10.20.

Листинг 10.20. Интерфейсы IReadable<T> и IIncrementable<T>

```
interface IReadable<T> {
    get(): T;
}

interface IIncrementable<T> {
    increment(): void;
}
```

В интерфейсе IReadable объявлен один метод, get(), извлекающий текущее значение из итератора

В интерфейсе IIncrementable объявлен один метод, increment(), переводящий итератор на следующий элемент

Эти два интерфейса — практически все, что требуется для поддержки наших исходных линейных алгоритмов обхода наподобие `map()`. Осталось только выяснить, в какой момент нужно останавливать работу. Нам известно, что сам итератор не знает, когда это делать, поскольку иногда не требуется обходить всю последовательность. Введем понятие равенства: итераторы `begin` и `end` равны, если указывают на один и тот же элемент. Это намного гибче стандартной реализации `Iterator<T>`. Можно задать начальное значение `end` на один элемент *после* финального элемента последовательности. А затем можно продвигать `begin` вперед, пока он не станет равен `end`, и в таком случае будет понятно, что мы обошли всю последовательность. Но можно также двигать итератор `end` назад, пока он не укажет на первый элемент последовательности, — то, что сделать с обычным `Iterator<T>` было нельзя (рис. 10.3).

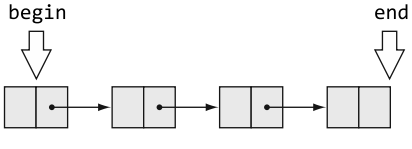


Рис. 10.3. Итераторы begin и end задают диапазон: begin указывает на первый его элемент, а end — на местоположение за последним элементом

В листинге 10.21 опишем интерфейс `IInputIterator<T>` как тот, который реализует оба интерфейса `IReadable<T>` и `IIncrementable<T>`, а также метод `equals()` для сравнения двух итераторов.

Листинг 10.21. Интерфейс IInputIterator<T>

```
interface IInputIterator<T> extends IReadable<T>, IIncrementable<T> {
    equals(other: IInputIterator<T>): boolean;
}
```

Сам итератор больше не может определить, произвел ли он обход последовательности полностью. Последовательность теперь задается двумя итераторами — одним, указывающим на начало последовательности, и вторым, указывающим на ее элемент, следующий за последним.

После описания этих интерфейсов мы в листинге 10.22 можем модифицировать наш итератор для связного списка из главы 9. Теперь данный список реализован в виде типа `LinkedListNode<T>` со свойствами `value` и `next`, причем последнее может принимать значение типа `LinkedListNode<T>` или `undefined`, в случае последнего узла в списке.

Листинг 10.22. Реализация связного списка

```
class LinkedListNode<T> {
    value: T;
    next: LinkedListNode<T> | undefined;

    constructor(value: T) {
        this.value = value;
    }
}
```

Смоделируем пару итераторов в целях обхода этого связного списка в листинге 10.23. Для начала необходимо реализовать `LinkedListInputIterator<T>`, который бы удовлетворял нашему новому контракту `IInputIterator<T>` для связного списка.

Листинг 10.23. Итератор ввода для связного списка

```
class LinkedListInputIterator<T> implements IInputIterator<T> {
    private node: LinkedListNode<T> | undefined;

    constructor(node: LinkedListNode<T> | undefined) {
        this.node = node;
    }

    increment(): void {
        if (!this.node) throw Error();

        this.node = this.node.next;
    }

    get(): T {
        if (!this.node) throw Error();

        return this.node.value;
    }

    equals(other: IInputIterator<T>): boolean {
        return this.node == (<LinkedListInputIterator<T>>other).node;
    }
}
```

Если свойство node текущего узла — undefined, то генерируем ошибку; в противном случае переходим на следующий узел

Если свойство node текущего узла — undefined, то генерируем ошибку; в ином случае получаем значение

Итераторы считаются равными, если служат обертками для одного узла. Приведение к типу LinkedListInputIterator<T> возможно, поскольку вызывающая сторона не должна сравнивать итераторы различных типов

Теперь можно создать пару итераторов для обхода связного списка, задав начальное значение `begin` равным голове списка, а `end` — равным `undefined`, как показано в листинге 10.24.

Листинг 10.24. Пара итераторов для обхода связного списка

```
const head: LinkedListNode<number> = new LinkedListNode(0);
head.next = new LinkedListNode(1);
head.next.next = new LinkedListNode(2);

let begin: IInputIterator<number> = new LinkedListInputIterator(head);
let end: IInputIterator<number> = new LinkedListInputIterator(undefined);
```

Список из нескольких узлов

begin указывает на переданную в качестве аргумента голову связного списка

end равен undefined

Это называется *итератором ввода* (input iterator), поскольку из него можно читать значения с помощью метода `get()`.

ИТЕРАТОРЫ ВВОДА

Итератор ввода — это итератор, способный обойти последовательность ровно один раз и вернуть ее значения. Выдать значения второй раз он не может, поскольку они могут оказаться недоступны. Итератор ввода не обязательно должен обходить постоянную структуру данных, он может выдавать значения из генератора или другого источника (рис. 10.4).

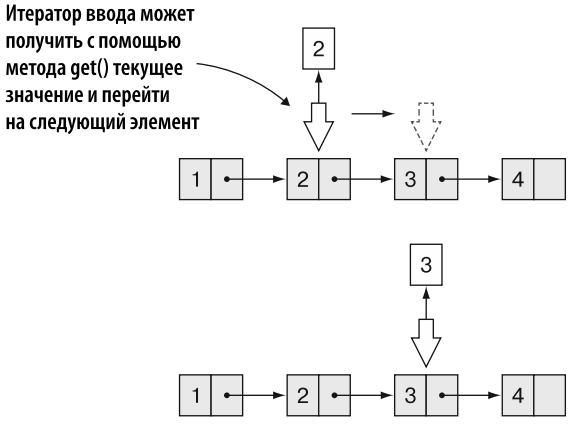


Рис. 10.4. Итератор ввода может извлечь значение текущего элемента и перейти к следующему

А теперь опишем итератор вывода — итератор, в который можно производить запись. Для этого объявим интерфейс `IWritable<T>`, включающий метод `set()`, и опишем наш `IOutputIterator<T>` как расширяющий `IWritable<T>`, `IIncrementable<T>` и включающий метод `equals()`, как показано в листинге 10.25.

Листинг 10.25. Интерфейсы IWritable<T> и IOutputIterator<T>

```
interface IWritable<T> {
    set(value: T): void;
}

interface IOutputIterator<T> extends IWritable<T>, IIncrementable<T> {
    equals(other: IOutputIterator<T>): boolean;
}
```

В подобный итератор можно записывать значения, но нельзя их оттуда прочитать.

ИТЕРАТОРЫ ВЫВОДА

> Итератор вывода (output iterator) — это итератор, который может произвести обход последовательности и записать в нее значения, но не обязан уметь читать их из нее. Он не обязательно должен обходить постоянную структуру данных; может также записывать значения в другие потоки вывода.

Реализуем итератор вывода для записи в консоль. Запись в поток вывода данных — самый распространенный сценарий использования итераторов вывода: вывести данные можно, а прочитать их обратно — нет. Мы можем записать данные (не имея возможности их прочитать снова) в сетевое соединение, стандартный поток вывода, стандартный поток ошибок и т. д. В нашем случае перевод итератора на следующую позицию никаких действий не означает, а операция установки значения приводит к вызову `console.log()`, как показано в листинге 10.26.

Листинг 10.26. Итератор вывода в консоль

```
class ConsoleOutputIterator<T> implements IOutputIterator<T> {
    set(value: T): void {
        console.log(value);
    }

    increment(): void {}

    equals(other: IOutputIterator<T>): boolean {
        return false;
    }
}
```

Метод set() выполняет запись в консоль

Метод increment() ничего не должен делать, так как в этом случае мы не обходим структуру данных

Метод equals() может всегда возвращать false, поскольку при записи в консоль отсутствует значение end для сравнения

Теперь в нашем распоряжении интерфейс, который описывает итератор ввода и конкретный экземпляр, реализующий обход нашего связного списка. Есть у нас и интерфейс, который описывает итератор вывода и конкретную его реализацию, записывающую информацию в консоль. С их помощью можно создать альтернативную реализацию `map()` в листинге 10.27.

Эта новая версия `map()` принимает в качестве аргумента пару итераторов ввода `begin` и `end`, задающих последовательность, и итератор вывода `out` для записи результатов отображения элементов последовательности с помощью заданной функции. А поскольку мы больше не используем стандартный синтаксис JavaScript,

не можем мы и применять часть его синтаксического сахара — оператор `yield` и циклы `for...of`.

Листинг 10.27. Алгоритм map() с итераторами ввода и вывода

```
function map<T, U>(
    begin: IInputIterator<T>, end: IInputIterator<T>,
    out: IOutputIterator<U>,
    func: (value: T) => U): void {

    while (!begin.equals(end)) {
        out.set(func(begin.get()));

        begin.increment();
        out.increment();
    }
}
```

Итераторы begin и end задают входную последовательность

out — итератор вывода для результатов работы функции

Повторяем, пока не обойдем всю последовательность и begin не станет равен end

Выводим результат применения функции к текущему элементу

Наращиваем значение итераторов ввода и вывода

Эта версия `map()` столь же универсальна, как и основанная на нативном интерфейсе `IterableIterator<T>`: в нее можно передать любой `IInputIterator<T>` — производящий обход связного списка, производящий центрированный обход дерева и т. д. Можно также передать любой `IOutputIterator<T>` — записывающий в консоль или в массив.

Пока это почти ничего нам не дало. Мы получили альтернативную реализацию, не способную использовать возможности специального синтаксиса TypeScript. Но эти итераторы — всего лишь базовые блоки. Можно описать итераторы с гораздо бо́льшими возможностями, чем мы и займемся далее.

10.4.2. Удобный алгоритм find()

Рассмотрим еще один распространенный алгоритм: `find()`. Он принимает на входе последовательность значений и предикат, а возвращает первый элемент последовательности, для которого предикат вернул `true`. Его можно реализовать с помощью стандартного `Iterable<T>`, как показано в листинге 10.28.

Листинг 10.28. Реализация алгоритма find() с итерируемым аргументом

```
function find<T>(iter: Iterable<T>,
    pred: (value: T) => boolean): T | undefined {
    for (const value of iter) {
        if (pred(value)) {
            return value;
        }
    }

    return undefined;
}
```

Эта реализация работает, но не особенно удобна. Что, если, найдя значение, мы захотим его поменять? Например, мы искали в связном списке первое вхождение числа 42, чтобы заменить его на 0. Чем нам поможет то, что `find()` вернет 42? С тем же успехом она могла возвращать `boolean`, ведь данная реализация просто сообщает нам, есть ли в последовательности такое значение.

Что, если вместо возврата самого значения итератор бы указывал на него? Готовый `Iterator<T>` JavaScript предназначен только для чтения. Но мы уже видели, как создать итератор, с помощью которого можно и устанавливать значения. Вышеописанный сценарий требует сочетания итератора для чтения и итератора для записи. Опишем так называемый прямой итератор.

ПРЯМЫЕ ИТЕРАТОРЫ

Прямой итератор[1] (forward iterator) — это итератор, который может продвигаться вперед, читать значение в текущей позиции и модифицировать его. Прямые итераторы можно также клонировать, вследствие чего перемещение вперед одной копии итератора не затрагивает остальные. Это важно, поскольку позволяет обходить последовательность несколько раз, в отличие от итераторов ввода и вывода (рис. 10.5).

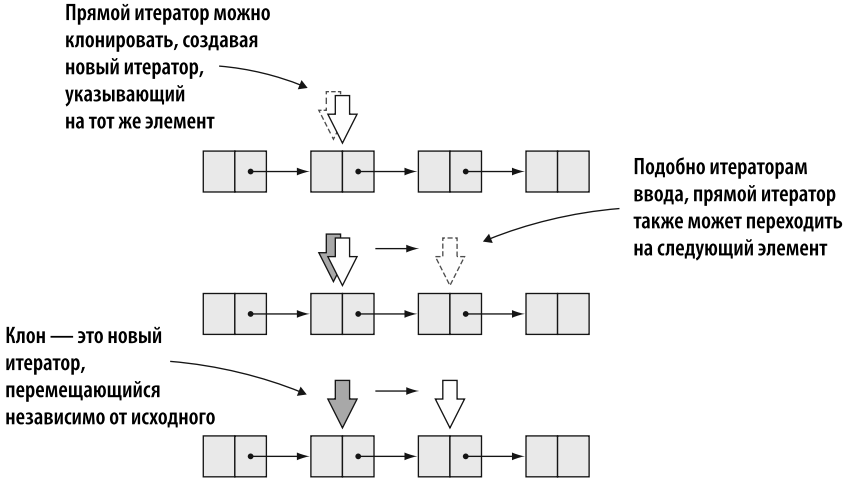


Рис. 10.5. Прямой итератор может читать и записывать значение текущего элемента, переходить к следующему и создавать свои копии, что позволяет многократно выполнять обход последовательностей. Здесь показано создание копии итератора с помощью метода clone(). При переводе исходного итератора на следующий элемент копия остается на той же позиции

Показанный в листинге 10.29 интерфейс `IForwardIterator<T>` представляет собой сочетание интерфейсов `IReadable<T>`, `IWritable<T>`, `IIncrementable<T>` и включает методы `equals()` и `clone()`.

[1] Встречается также под названиями «поступательный итератор», «однонаправленный итератор». — *Примеч. пер.*

Листинг 10.29. Интерфейс IForwardIterator<T>

```
interface IForwardIterator<T> extends
    IReadable<T>, IWritable<T>, IIncrementable<T> {
    equals(other: IForwardIterator<T>): boolean;
    clone(): IForwardIterator<T>;
}
```

В качестве примера реализуем данный интерфейс для обхода связного списка в листинге 10.30. Модифицируем класс `LinkedListIterator<T>`, введя в него требуемые этим новым интерфейсом дополнительные методы.

Листинг 10.30. Класс LinkedListIterator<T>, реализующий интерфейс IForwardIterator<T>

```
class LinkedListIterator<T> implements IForwardIterator<T> {
    private node: LinkedListNode<T> | undefined;

    constructor(node: LinkedListNode<T> | undefined) {
        this.node = node;
    }

    increment(): void {
        if (!this.node) return;
        this.node = this.node.next;
    }

    get(): T {
        if (!this.node) throw Error();

        return this.node.value;
    }

    set(value: T): void {
        if (!this.node) throw Error();

        this.node.value = value;
    }

    equals(other: IForwardIterator<T>): boolean {
        return this.node == (<LinkedListIterator<T>>other).node;
    }

    clone(): IForwardIterator<T> {
        return new LinkedListIterator(this.node);
    }
}
```

Эта версия класса LinkedListIterator<T> реализует новый интерфейс IForwardIterator<T>

Дополнительный метод set(), требуемый интерфейсом IWritable<T>, служит для модификации значения узла связного списка

Метод equals() теперь ожидает в качестве параметра IForwardIterator<T>

Метод clone() создает новый итератор, указывающий на тот же узел, что и итератор this

Теперь посмотрим на приведенную в листинге 10.31 версию функции `find()`, которая принимает в качестве аргументов пару итераторов `begin` и `end` и возвращает итератор, который указывает на первый удовлетворяющий предикату элемент. Использование этой версии позволяет обновлять найденное значение.

Воспользуемся только что реализованным нами итератором для обхода связного списка чисел, найдем с помощью этого алгоритма первое значение, равное `42`, и заменим его на `0`, как показано в листинге 10.32.

Листинг 10.31. Функция find() с прямым итератором

```
function find<T>(
    begin: IForwardIterator<T>, end: IForwardIterator<T>,
    pred: (value: T) => boolean): IForwardIterator<T> {
    while (!begin.equals(end)) {
        if (pred(begin.get())) {
            return begin;
        }

        begin.increment();
    }

    return end;
}
```

Повторяем, пока не обойдем всю последовательность

Прямые итераторы begin и end задают последовательность

Наша функция возвращает прямой итератор, указывающий на найденный элемент

Если искомый элемент найден, то возвращаем итератор

Переходим к очередному элементу последовательности

Если мы добрались до end, то элемент не найден. Возвращаем итератор end

Листинг 10.32. Замена 42 на 0 в связном списке

```
let head: LinkedListNode<number> = new LinkedListNode(1);
head.next = new LinkedListNode(2);
head.next.next = new LinkedListNode(42);

let begin: IForwardIterator<number> =
    new LinkedListIterator(head);
let end: IForwardIterator<number> =
    new LinkedListIterator(undefined);

let iter: IForwardIterator<number> =
    find(begin, end, (value: number) => value == 42);

if (!iter.equals(end)) {
    iter.set(0);
}
```

Создаем связный список, содержащий последовательность 1, 2, 42

Задаем начальные значения прямых итераторов begin и end для связного списка

Вызываем функцию find() и получаем итератор, указывающий на первый узел, содержащий значение 42

Необходимо убедиться, что мы нашли узел со значением 42, а не дошли до конца списка

Если нашли, то обновляем его значение на 0

Прямые итераторы способны на очень многое: обходить последовательность произвольное количество раз и модифицировать ее. С их помощью можно реализовывать алгоритмы, не требующие создания дополнительных структур данных, которым не нужно копировать всю последовательность данных, чтобы ее преобразовать. Наконец займемся алгоритмом, с которого мы начали этот раздел: `reverse()`.

10.4.3. Эффективная реализация reverse()

Как мы видели в реализации на основе массива, не требующая создания дополнительных структур данных реализация `reverse()` начинает работу с обоих концов массива одновременно и меняет элементы местами, наращивая значение прямого индекса и уменьшая значение обратного до тех пор, пока они не совпадут.

Можно обобщить реализацию на основе массива для работы с произвольной последовательностью, но нашему итератору для этого понадобится еще одно:

способность переходить на предыдущий элемент. Обладающий ею итератор называется *двунаправленным* (bidirectional iterator).

ДВУНАПРАВЛЕННЫЙ ИТЕРАТОР

Двунаправленный итератор может все то же, что и прямой, и вдобавок способен переходить на предыдущий элемент. Другими словами, двунаправленный итератор может перемещаться по последовательности как вперед, так и назад (рис. 10.6).

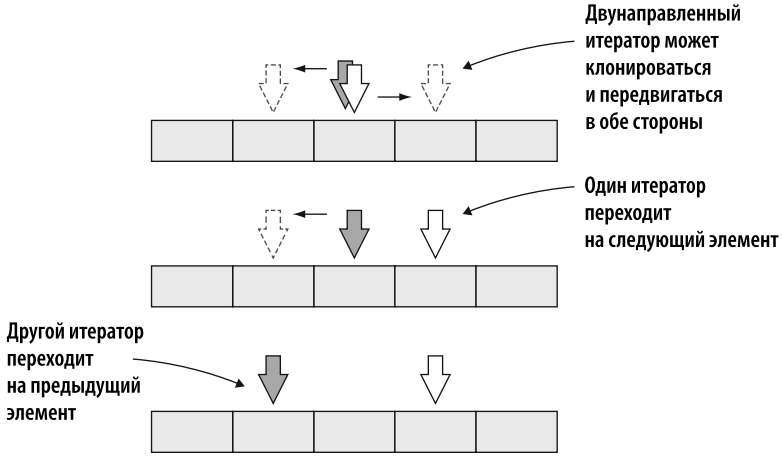


Рис. 10.6. Двунаправленный итератор способен читать/записывать значение текущего элемента, клонировать себя, а также перемещаться вперед и назад по последовательности

Опишем интерфейс `IBidirectionalIterator<T>`, аналогичный интерфейсу `IForwardIterator<T>`, но с дополнительным методом `decrement()`. Обратите внимание: далеко не все структуры данных, например наш связный список, поддерживают подобный итератор. Поскольку в связном списке у каждого элемента есть ссылка только на следующий узел, перейти обратно к предыдущему узлу не получится. Но можно создать двунаправленный итератор для двусвязного списка, в котором каждый узел включает ссылки на предыдущий и последующий узлы, или для массива. Реализуем `IBidirectionalIterator<T>` в виде класса `ArrayIterator<T>` в листинге 10.33.

Листинг 10.33. Интерфейс IBidirectionalIterator<T> и класс ArrayIterator<T>

```
interface IBidirectionalIterator<T> extends
    IReadable<T>, IWritable<T>, IIncrementable<T> {
    decrement(): void;
    equals(other: IBidirectionalIterator<T>): boolean;
    clone(): IBidirectionalIterator<T>;
}

class ArrayIterator<T> implements IBidirectionalIterator<T> {
    private array: T[];
    private index: number;
```

В интерфейс IBidirectionalIterator<T>, по сравнению с IForwardIterator<T>, включен дополнительный метод decrement()

```
    constructor(array: T[], index: number) {
        this.array = array;
        this.index = index;
    }

    get(): T {
        return this.array[this.index];
    }

    set(value: T): void {
        this.array[this.index] = value;
    }

    increment(): void {
        this.index++;
    }

    decrement(): void {
        this.index--;
    }

    equals(other: IBidirectionalIterator<T>): boolean {
        return this.index == (<ArrayIterator<T>>other).index;
    }

    clone(): IBidirectionalIterator<T> {
        return new ArrayIterator(this.array, this.index);
    }
}
```

Теперь реализуем алгоритм `reverse()` с помощью пары двунаправленных итераторов `begin` и `end` (листинг 10.34). Меняем значения местами, передвигаем `begin` вперед, `end` назад и завершаем работу, когда эти два итератора совпадают. При этом важно гарантировать, что наши итераторы не «проскочат» друг друга, поэтому сразу после перемещения одного из них необходимо проверить, не совпали ли они.

Листинг 10.34. Реализация алгоритма reverse() с помощью двунаправленных итераторов

```
function reverse<T>(
    begin: IBidirectionalIterator<T>, end: IBidirectionalIterator<T>
    ): void {
    while (!begin.equals(end)) {
        end.decrement();
        if (begin.equals(end)) return;

        const temp: T = begin.get();
        begin.set(end.get());
        end.set(temp);

        begin.increment();
    }
}
```

Повторяем до тех пор, пока итераторы begin и end не совпадут

Передвигаем end назад. Напомним: end начинаем со следующего за концом массива значения, поэтому его нужно передвинуть назад, прежде чем использовать

Меняем значения местами

Снова проверяем, что в результате передвижения end назад два итератора не стали указывать на один элемент

Наконец, передвигаем begin вперед и повторяем итерацию цикла (в условии цикла while снова проверяется, не совпали ли наши два итератора)

Попробуем его на деле в листинге 10.35.

Листинг 10.35. Обращение массива чисел

```
let array: number[] = [1, 2, 3, 4, 5];

let begin: IBidirectionalIterator<number>
    = new ArrayIterator(array, 0);
let end: IBidirectionalIterator<number>
    = new ArrayIterator(array, array.length);

reverse(begin, end);

console.log(array);
```

Инициализируем begin итератором для обхода массива, указывающим на позицию 0

Инициализируем end итератором для обхода массива, указывающим на элемент с индексом length (следующий за последним элементом массива)

В результате в консоль выводится [5, 4, 3, 2, 1]

Двунаправленные итераторы позволяют обобщить эффективный, работающий без создания дополнительных структур данных алгоритм `reverse()` на произвольные структуры данных, допускающие обход в двух направлениях. Мы расширили исходный алгоритм, возможности которого ограничивались работой с массивами, до работы с любым `IBidirectionalIterator<T>`. Один и тот же алгоритм теперь пригоден для обращения двусвязного списка или любой другой структуры данных, по которой итератор можно перемещать вперед и назад.

Обратите внимание: можно реализовать и алгоритм обращения односвязного списка, но обобщить такой алгоритм не получится. При обращении односвязного списка приходится вносить изменения в структуру данных, заменяя ссылки на следующие элементы ссылками на предыдущие. Подобный алгоритм очень тесно привязан к конкретной структуре данных и не подлежит обобщению. А наш требующий двунаправленного итератора обобщенный `reverse()`, напротив, с успехом работает для любой структуры данных, которая может обеспечить подобный итератор.

10.4.4. Эффективное извлечение элементов

Существуют алгоритмы, требующие от итераторов намного больше, чем простые операции `increment()` и `decrement()`. Хороший пример: алгоритмы сортировки. Эффективному алгоритму сортировки, работающему за время $O(n \log n)$, такому как быстрая сортировка, приходится «прыгать» по сортируемой структуре данных вперед и назад, обращаясь к элементам в произвольных местах. Двунаправленного итератора для этого недостаточно. Нам понадобится итератор произвольного доступа.

ИТЕРАТОРЫ ПРОИЗВОЛЬНОГО ДОСТУПА

Итератор произвольного доступа (random-access iterator) может перемещаться на любое заданное количество элементов вперед/назад за константное время. В отличие от двунаправленного, способного перемещаться вперед/назад только на один элемент за раз, итератор произвольного доступа может перемещаться на любое количество элементов (рис. 10.7).

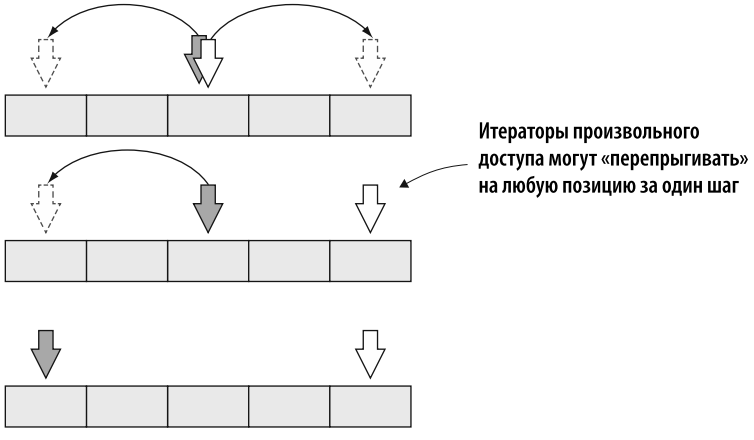


Рис. 10.7. Итератор произвольного доступа способен читать/записывать значение текущего элемента, клонироваться, а также перемещаться вперед или назад на любое количество элементов

Массив — хороший пример структуры данных с произвольным доступом, в нем можно быстро обратиться к любому элементу по индексу и извлечь его. И напротив, в случае двусвязного списка необходимо пройти по всем соответствующим ссылкам на последующий/предыдущий элемент, чтобы достичь нужного элемента. Двусвязные списки не поддерживают итераторы произвольного доступа.

Опишем `IRandomAccessIterator<T>` — итератор, который не только поддерживает все возможности `IBidirectionalIterator<T>`, но и включает метод `move()`, перемещающий итератор на *n* элементов. При работе с итераторами произвольного доступа не помешает также знать, насколько далеко друг от друга находятся два итератора. Поэтому в листинге 10.36 мы добавим метод `distance()`, который возвращает расстояние между двумя итераторами.

Листинг 10.36. Итератор IRandomAccessIterator<T>

```
interface IRandomAccessIterator<T>
    extends IReadable<T>, IWritable<T>, IIncrementable<T> {
    decrement(): void;
    equals(other: IRandomAccessIterator<T>): boolean;
    clone(): IRandomAccessIterator<T>;
    move(n: number): void;
    distance(other: IRandomAccessIterator<T>): number;
}
```

В листинге 10.37 модифицируем наш `ArrayIterator<T>` так, чтобы он реализовывал интерфейс `IRandomAccessIterator<T>`.

Рассмотрим очень простой алгоритм, для которого может пригодиться итератор произвольного доступа: `elementAt()`. Данный алгоритм принимает в качестве аргументов итераторы `begin` и `end`, задающие последовательность и число *n*, а возвращает

итератор, указывающий на n-й элемент последовательности или итератор end, если n превышает длину последовательности.

Листинг 10.37. Класс ArrayIterator<T>, реализующий итератор произвольного доступа

```
class ArrayIterator<T> implements IRandomAccessIterator<T> {
    private array: T[];
    private index: number;

    constructor(array: T[], index: number) {
        this.array = array;
        this.index = index;
    }

    get(): T {
        return this.array[this.index];
    }

    set(value: T): void {
        this.array[this.index] = value;
    }

    increment(): void {
        this.index++;
    }

    decrement(): void {
        this.index--;
    }

    equals(other: IRandomAccessIterator<T>): boolean {
        return this.index == (<ArrayIterator<T>>other).index;
    }

    clone(): IRandomAccessIterator<T> {
        return new ArrayIterator(this.array, this.index);
    }

    move(n: number): void {
        this.index += n;
    }

    distance(other: IRandomAccessIterator<T>): number {
        return this.index - (<ArrayIterator<T>>other).index;
    }
}
```

Метод move() перемещает итератор на n элементов (n может быть отрицательным для перемещения в обратном направлении)

Метод distance() вычисляет расстояние между двумя итераторами

Данный алгоритм можно реализовать с помощью итератора ввода, но его пришлось бы перемещать n раз, чтобы достичь нужного элемента. А это означает линейную сложность алгоритма, то есть $O(n)$. Итератор произвольного доступа позволяет сделать то же самое за константное время, $O(1)$, как показано в листинге 10.38.

Листинг 10.38. Получение элемента, расположенного на заданной позиции

```
function elementAtRandomAccessIterator<T>(
    begin: IRandomAccessIterator<T>, end: IRandomAccessIterator<T>,
    n: number): IRandomAccessIterator<T> {
    begin.move(n);

    if (begin.distance(end) <= 0) return end;

    return begin;
}
```

Перемещает итератор begin на n элементов вперед

Если он равен или больше end, то n превышает длину последовательности, так что возвращаем end

В ином случае возвращаем итератор, указывающий на нужный элемент

С помощью итераторов произвольного доступа можно реализовать наиболее эффективные алгоритмы, но очень немногие структуры данных могут обеспечить работу таких итераторов.

10.4.5. Краткое резюме по итераторам

Мы рассмотрели разнообразные категории итераторов и создание на основе их различных возможностей более эффективных алгоритмов. Мы начали с итераторов ввода и вывода, предназначенных для одноразового обхода последовательности. Итераторы ввода позволяют читать значения, а итераторы вывода — задавать их.

Этого вполне достаточно для таких алгоритмов, как `map()`, `filter()` и `reduce()`, обрабатывающих входные данные последовательно. В большинстве языков программирования, включая Java и C# с их интерфейсами `Iterable<T>` и `IEnumerable<T>`, существуют библиотеки только для данной разновидности итераторов.

Далее мы видели, каким образом способность итераторов как читать, так и записывать значения вкупе с созданием копии итератора позволяет реализовывать другие удобные алгоритмы, модифицирующие данные, не прибегая к созданию дополнительных структур. Этими новыми возможностями обладают так называемые прямые итераторы.

В некоторых случаях, например в примере с алгоритмом `reverse()`, недостаточно иметь возможность перемещаться по последовательности только вперед. Необходимо перемещаться в обе стороны. Итератор, способный перемещаться как вперед, так и назад, называется двунаправленным.

Наконец, на эффективности работы некоторых алгоритмов положительно сказывается возможность «прыгать» по последовательности и обращаться к элементам в произвольных местах, а не проходить последовательность шаг за шагом. Хороший пример — алгоритмы сортировки; таков и только что обсуждавшийся простой алгоритм `elementAt()`. Для поддержки подобных алгоритмов используются итераторы произвольного доступа, способные перемещаться на несколько элементов за раз.

Эти идеи не новы; стандартная библиотека C++ включает набор эффективных алгоритмов, в которых применяются итераторы со схожими возможностями. Прочие языки ограничиваются меньшим набором алгоритмов или менее эффективными реализациями.

Возможно, вы обратили внимание, что основанные на итераторах алгоритмы не текучие, поскольку принимают на входе пару итераторов и возвращают `void` или другой итератор. C++ сейчас переходит от итераторов к использованию интервалов (ranges). Мы не станем подробно обсуждать этот вопрос в этой книге, но в общих чертах интервал можно считать парой итераторов `begin/end`. Модификация алгоритмов для получения интервалов в качестве аргументов и возврата интервалов позволяет создавать более текучие API, сцепляя операции над интервалами. Вероятно, в будущем алгоритмы на основе интервалов будут реализованы и в других языках программирования. Возможность выполнять эффективные обобщенные алгоритмы над любой структурой данных, не прибегая к созданию дополнительных структур данных с помощью достаточно функциональных итераторов, исключительно полезна.

10.4.6. Упражнения

1. Какова минимальная категория итератора, достаточная для реализации алгоритма `drop()`, пропускающего первые n элементов диапазона?
   А. Итератор ввода.
   Б. Прямой итератор.
   В. Двунаправленный итератор.
   Г. Итератор с произвольным доступом.
2. Какова минимальная категория итератора, достаточная для реализации алгоритма бинарного поиска (со сложностью $O(\log n)$)? Напомню, что при бинарном поиске проверяется средний элемент диапазона[1]. Если он больше искомого значения, то диапазон разбивается напополам и далее анализируется первая половина. Если нет, то анализируется вторая половина диапазона, после чего процедура повторяется. А поскольку область поиска уменьшается вдвое на каждом шаге, то сложность алгоритма — $O(\log n)$.
   А. Итератор ввода.
   Б. Прямой итератор.
   В. Двунаправленный итератор.
   Г. Итератор с произвольным доступом.

10.5. Адаптивные алгоритмы

Чем больше требований к итератору, тем меньшее количество структур данных может его поддерживать. Мы видели, что можно создать прямой итератор для односвязного списка, двусвязного или массива. Если нам требуется двунаправленный итератор, то односвязные списки выбывают из состязания. Можно создать двунаправленный итератор для обхода двусвязных списков и массивов, но не одно-

[1] Отсортированного в порядке возрастания. — *Примеч. пер.*

связных списков. А для итераторов с произвольным доступом не подходят уже даже двусвязные списки.

Хочется, чтобы обобщенные алгоритмы были как можно более обобщенными. Для этого требуются итераторы с минимально необходимыми для данного алгоритма возможностями. Но, как мы только что видели, требования менее эффективных версий алгоритмов к итераторам не столь высоки. Можно создать несколько версий некоторых алгоритмов: менее эффективную и более эффективную, работающие с соответствующе продвинутыми итераторами.

Вернемся к нашему примеру `elementAt()`. Этот алгоритм возвращает n-й элемент последовательности или ее конец, если n больше ее длины. Чтобы использовать прямой итератор, его можно переместить вперед n раз и вернуть значение. Сложность алгоритма при этом линейная, $O(n)$, так как количество шагов растет пропорционально n. С другой стороны, с помощью итератора произвольного доступа можно извлечь элемент за константное время $O(1)$.

Что лучше: более общий и менее эффективный алгоритм или более эффективный, но ограниченный меньшим кругом структур данных? На самом деле выбирать вовсе не требуется: можно создать две версии алгоритма и использовать наиболее подходящую реализацию в зависимости от доступного типа итератора.

Реализуем `elementAtForwardIterator()` и `elementAtRandomAccessIterator()`, извлекающие элемент за линейное и константное время соответственно, как показано в листинге 10.39.

Листинг 10.39. Реализация elementAt() с помощью итератора ввода и итератора произвольного доступа

```
function elementAtForwardIterator<T>(
    begin: IForwardIterator<T>, end: IForwardIterator<T>,
    n: number): IForwardIterator<T> {
    while (!begin.equals(end) && n > 0) {
        begin.increment();
        n--;
    }

    return begin;
}

function elementAtRandomAccessIterator<T>(
    begin: IRandomAccessIterator<T>, end: IRandomAccessIterator<T>,
    n: number): IRandomAccessIterator<T> {
    begin.move(n);

    if (begin.distance(end) <= 0) return end;

    return begin;
}
```

Если n больше 0 и конец последовательности не достигнут, то перемещаем итератор на следующий элемент и уменьшаем n на единицу

Возвращаем итератор begin. Он указывает либо на n-й элемент последовательности, либо на ее конец

Это реализация алгоритма elementAt() из предыдущего раздела

Теперь можно реализовать функцию `elementAt()`, которая будет выбирать алгоритм в зависимости от возможностей, полученных в качестве аргументов итераторов, как показано в листинге 10.40. Обратите внимание: TypeScript не поддерживает

перегрузку функций, поэтому нам понадобится функция для определения типа итератора. В других языках программирования, например C# и Java, можно просто создать методы с одним названием, но разными аргументами.

Листинг 10.40. Адаптивный алгоритм elementAt()

```
function isRandomAccessIterator<T>(
    iter: IForwardIterator<T>): iter is IRandomAccessIterator<T> {
    return "distance" in iter;
}

function elementAt<T>(
    begin: IForwardIterator<T>, end: IForwardIterator<T>,
    n: number): IForwardIterator<T> {
    if (isRandomAccessIterator(begin) && isRandomAccessIterator(end)) {
        return elementAtRandomAccessIterator(begin, end, n);
    } else {
        return elementAtForwardIterator(begin, end, n);
    }
}
```

Считаем, что iter — итератор произвольного доступа, если у него есть метод distance

Если итераторы произвольного доступа, то вызываем эффективную функцию elementAtRandomAccessIterator()

Если нет, то обращаемся к менее эффективной функции elementAtForwardIterator()

Хороший алгоритм использует имеющиеся возможности, адаптируясь к менее продвинутому итератору путем использования менее эффективной реализации, а для более продвинутых итераторов приберегая более эффективную.

10.5.1. Упражнение

Реализуйте `nthLast()` — функцию, которая возвращает итератор, указывающий на n-й с конца элемент диапазона (или `end`, если диапазон слишком узок). Если $n = 1$, то возвращаем итератор, указывающий на последний элемент; если $n = 2$, то итератор, указывающий на предпоследний, и т. д. Если $n = 0$, то возвращаем итератор `end`, который указывает на следующий за последним элемент диапазона.

Подсказка: ее можно реализовать с помощью `ForwardIterator` с двумя проходами. При первом проходе мы подсчитываем элементы диапазона. На втором проходе мы уже знаем размер диапазона, а значит, знаем, когда остановиться, чтобы оказаться за n элементов до его конца.

Резюме

- Обобщенные алгоритмы работают с итераторами, что позволяет повторно использовать их для различных структур данных.
- Вместо того чтобы писать цикл, задумайтесь, нельзя ли решить свою задачу с помощью библиотечного алгоритма или сочетания алгоритмов.
- Текучие API обеспечивают удобный интерфейс для сцепления алгоритмов.

- Ограничения типов позволяют алгоритмам выдвигать определенные требования к типам, с которыми они работают.
- Итераторы ввода могут читать значения и перемещаться вперед по последовательности на один элемент за шаг. Можно читать из потока данных, например стандартного потока ввода, с помощью входных итераторов. Перечитать уже прочитанное значение еще раз нельзя, можно только переместиться вперед.
- Итераторы вывода позволяют записывать значения и перемещаться вперед по последовательности на один элемент за шаг. Можно производить запись в поток данных, например в стандартный поток ввода, с помощью итераторов вывода. Прочитать уже записанное значение нельзя.
- Прямые итераторы могут читать и записывать значения, их можно перемещать вперед по последовательности и клонировать. В качестве хорошего примера структуры данных, поддерживающей прямые итераторы, можно привести связный список. В этой структуре можно перейти к следующему элементу и хранить несколько ссылок на текущий элемент, но нельзя перейти к предыдущему, если не сохранить ссылку на него, изначально находясь на нем.
- Двунаправленные итераторы обладают всеми возможностями прямых, но могут еще и перемещаться в обратном направлении. В качестве примера структуры данных, поддерживающей двунаправленные итераторы, можно привести двусвязный список. При необходимости в нем можно перейти как к следующему, так и предыдущему элементу.
- Итераторы с произвольным доступом могут свободно перемещаться за один шаг на любую позицию в последовательности. Одна из структур данных, поддерживающая итераторы с произвольным доступом, — массив. В нем можно «перепрыгнуть» за один шаг к любому элементу.
- В большинство основных языков программирования включены библиотеки алгоритмов для итераторов ввода.
- Чем больше возможностей у итератора, тем более эффективные алгоритмы можно создать с его помощью.
- Адаптивные алгоритмы включают несколько реализаций: чем большими возможностями обладает итератор, тем эффективнее работает алгоритм.

В главе 11 мы поднимемся на следующий уровень абстракции — к типам, принадлежащим к более высокому роду, — и узнаем, что такое монады и какие возможности доступны благодаря им.

Ответы к упражнениям

10.1. Улучшенные операции map(), filter() и reduce()

1. Одна из возможных реализаций на основе `reduce()` и `filter()`:

```
function concatenateNonEmpty(iter: Iterable<string>): string {
    return reduce(
        filter(
```

```
            iter,
            (value) => value.length > 0),
        "", (str1: string, str2: string) => str1 + str2);
}
```

2. Одна из возможных реализаций на основе `map()` и `filter()`:

```
function squareOdds(iter: Iterable<number>): IterableIterator<number> {
    return map(
        filter(
            iter,
            (value) => value % 2 == 1),
        (x) => x * x
        );
}
```

10.2. Распространенные алгоритмы

1. Одна из возможных реализаций:

```
class FluentIterable<T> {
    /* ... */

    take(n: number): FluentIterable<T> {
        return new FluentIterable (this.takeImpl(n));
    }

    private *takeImpl(n: number): IterableIterator<T> {
        for (const value of this.iter) {
            if (n-- <= 0) return;

            yield value;
         }
    }
}
```

2. Одна из возможных реализаций:

```
class FluentIterable<T> {
    /* ... */

    drop(n: number): FluentIterable<T> {
        return new FluentIterable(this.dropImpl(n));
    }

    private *dropImpl(n: number): IterableIterator<T> {
        for (const value of this.iter) {
            if (n-- > 0) continue;

            yield value;
        }
    }
}
```

10.3. Ограничение типов-параметров

1. Одно из возможных решений, использующее ограничение обобщенного типа, чтобы гарантировать, что `T` является разновидностью `IComparable`:

```
function clamp<T extends IComparable<T>>(value: T, low: T, high: T): T {
    if (value.compareTo(low) == ComparisonResult.LessThan) {
        return low;
    }

    if (value.compareTo(high) == ComparisonResult.GreaterThan) {
        return high;
    }

    return value;
}
```

10.4. Эффективная реализация reverse и других алгоритмов с помощью итераторов

1. А — `drop()` подойдет даже для потенциально бесконечных потоков данных. Возможности перемещаться вперед по последовательности вполне достаточно.
2. Г — для эффективного бинарного поиска необходима возможность «перепрыгивать» на каждом шаге в середину диапазона. Даже двунаправленному итератору пришлось бы проходить по последовательности элемент за элементом, чтобы достичь середины диапазона, поэтому сложности $O(\log n)$ алгоритм не достиг бы. (Подобный проход шаг за шагом дает $O(n)$, то есть линейное время выполнения.)

10.5. Адаптивные алгоритмы

Адаптивный алгоритм должен перемещаться в обратном направлении, если получает на входе двунаправленные итераторы, либо использовать подход с двумя проходами, если получает прямые. Вот одна из возможных реализаций:

```
function nthLastForwardIterator<T>(
    begin: IForwardIterator<T>, end: IForwardIterator<T>, n: number)
    : IForwardIterator<T> {
    let length: number = 0;
    let begin2: IForwardIterator<T> = begin.clone();

    // определяем длину интервала
    while (!begin.equals(end)) {
        begin.increment();
        length++;
    }

    if (length < n) return end;

    let curr: number = 0;
```

```
    // перемещаем итератор вперед, пока n-й элемент с конца
    // последовательности не окажется текущим
    while (!begin2.equals(end) && curr < length - n) {
        begin2.increment();
        curr++;
    }

    return begin2;
}

function nthLastBidirectionalIterator<T>(
    begin: IBidirectionalIterator<T>, end: IBidirectionalIterator<T>,
n: number)
    : IBidirectionalIterator<T> {
    let curr: IBidirectionalIterator<T> = end.clone();

    while (n > 0 && !curr.equals(begin)) {
        curr.decrement();
        n--;
    }

    // если мы достигли begin до того, как переместили итератор
    // назад n раз, то, значит, интервал слишком мал
    if (n > 0) return end;

    return curr;
}

function isBidirectionalIterator<T>(
    iter: IForwardIterator<T>): iter is IBidirectionalIterator<T> {
    return "decrement" in iter;
}

function nthLast<T>(
    begin: IForwardIterator<T>, end: IForwardIterator<T>, n: number)
    : IForwardIterator<T> {
    if (isBidirectionalIterator(begin) && isBidirectionalIterator(end))
{
        return nthLastBidirectionalIterator(begin, end, n);
    } else {
        return nthLastForwardIterator(begin, end, n);
    }
}
```

11 Типы, относящиеся к более высокому роду, и не только

В этой главе

- Применение операции `map()` к прочим типам.
- Инкапсуляция распространения ошибки.
- Монады и их приложения.
- Источники информации для дальнейшего изучения.

На протяжении этой книги мы видели разнообразные версии очень распространенного алгоритма `map()`, а в главе 10 наблюдали итераторы — абстракцию, с помощью которой можно повторно использовать его для разнообразных структур данных. В текущей главе мы увидим, как выйти за пределы итераторов и создать еще более общую версию этого замечательного алгоритма. Он позволит нам комбинировать обобщенные типы и функции и обеспечит единый способ обработки ошибок.

После изучения нескольких примеров я приведу определение этого широко применимого семейства функций, известных под названием функторов. Также будет рассказано, что такое типы, относящиеся к более высокому роду, и как с их помощью описывать подобные обобщенные функции. Рассмотрим ограничения, возникающие при использовании языков программирования, не поддерживающие типы, относящиеся к более высокому роду.

Затем мы обсудим монады. Этот термин встречается достаточно часто, и, хотя на первый взгляд выглядит устрашающе, идея очень проста. Мы узнаем, что такое

монады, и рассмотрим несколько их приложений, начиная с усовершенствованного механизма передачи ошибки далее и заканчивая асинхронным кодом и схлопыванием последовательностей.

И в завершение этой главы вас ждет раздел, в котором мы будем обсуждать некоторые из вопросов, уже встречавшихся в данной книге, и еще несколько разновидностей типов, не представленных мной: зависимые и линейные типы данных. Я не стану вдаваться в подробности, а приведу только краткое резюме и перечислю несколько дополнительных источников информации на случай, если вы захотите узнать о них подробнее. Я порекомендую несколько книг, содержащих больше информации по каждой из этих тем, а также назову языки программирования, поддерживающие некоторые из этих возможностей.

11.1. Еще более обобщенная версия алгоритма map

В главе 10 мы усовершенствовали реализацию алгоритма `map()` из главы 5, работавшую только с массивами, до приведенной в листинге 11.1 обобщенной реализации, работающей на основе итераторов. Мы выяснили, как итераторы способны абстрагировать логику обхода структуры данных, поэтому наша новая версия `map()` могла применять заданную функцию к элементам произвольной структуры (рис. 11.1).

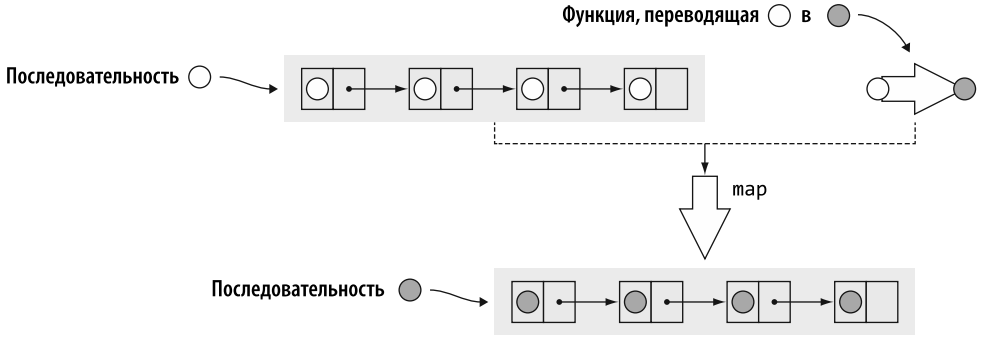


Рис. 11.1. Алгоритм map() принимает на входе итератор обхода последовательности, в данном случае списка кругов, и функцию преобразования круга, применяет эту функцию к каждому из элементов последовательности и генерирует новую последовательность с измененными элементами

Листинг 11.1. Обобщенный алгоритм map()

```
function* map<T, U>(iter: Iterable<T>, func: (item: T) => U):
    IterableIterator<U> {
    for (const value of iter) {
        yield func(value);
    }
}
```

Эта реализация может работать с итераторами, но нам хотелось бы иметь возможность применять функцию вида `(item: T) => U` и к другим типам данных. В качестве примера возьмем описанный в главе 3 тип `Optional<T>`, приведенный в листинге 11.2.

Листинг 11.2. Тип Optional

```
class Optional<T> {
    private value: T | undefined;
    private assigned: boolean;

    constructor(value?: T) {
        if (value) {
            this.value = value;
            this.assigned = true;
        } else {
            this.value = undefined;
            this.assigned = false;
        }
    }

    hasValue(): boolean {
        return this.assigned;
    }

    getValue(): T {
        if (!this.assigned) throw Error();

        return <T>this.value;
    }
}
```

Отображение `Optional<T>` с помощью функции `(value: T) => U` выглядит вполне естественным. Если опционал содержит значение типа `T`, то отображение его с помощью функции должно дать `Optional<U>`, содержащий результат применения этой функции. С другой стороны, при пустом опционале в результате отображения должен быть возвращен пустой `Optional<U>` (рис. 11.2).

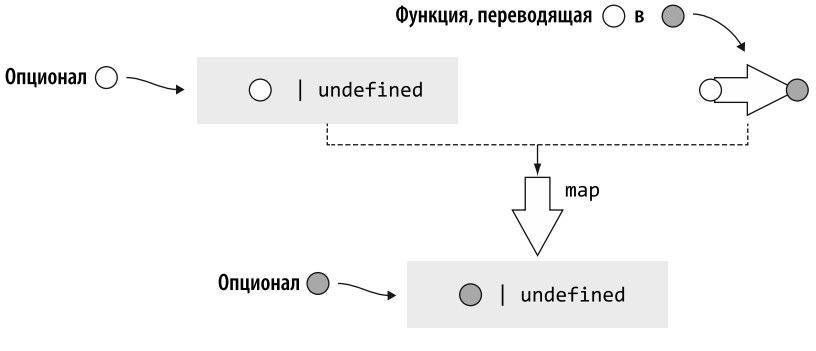


Рис. 11.2. Отображение опционального значения с помощью функции. Если опционал пуст, то map() возвращает пустой опционал; в противном случае функция применяется к значению и возвращается опционал с результатом

Создадим набросок реализации данной версии map. Поместим эту функцию в пространство имен (листинг 11.3). Поскольку TypeScript не поддерживает перегрузки функций, несколько функций с одним названием необходимо размещать в разных пространствах имен, чтобы компилятор мог определить, какую из них мы вызываем.

Листинг 11.3. Опциональный алгоритм map()

```
namespace Optional {
    export function map<T, U>(
        optional: Optional<T>, func: (value: T) => U): Optional<U> {
        if (optional.hasValue()) {
            return new Optional<U>(func(optional.getValue()));
        }else {
            return new Optional<U>();
        }
    }
}
```

Оператор export обеспечивает видимость функции вне ее пространства имен

Если опционал содержит значение, то извлекаем его, передаем func() и инициализируем Optional<U> полученным в результате ее выполнения значением

Если опционал пуст, то создаем новый пустой Optional<U>

Нечто подобное можно сделать и с типом-суммой T или undefined языка TypeScript. Напомню, что Optional<T> — наша самодельная версия подобного типа, работающая даже в языках, в которых нет нативной поддержки типов-сумм, но в TypeScript она присутствует. Посмотрим на нативное отображение опционального типа данных T | undefined (листинг 11.4).

Отображение T | undefined с помощью функции (value: T) => U означает применение функции и возврат ее результата в случае значения типа T или возврат undefined, если это значение было начальным.

Листинг 11.4. Алгоритм map() для типа-суммы

```
namespace SumType {
    export function map<T, U>(
        value: T | undefined, func: (value: T) => U): U | undefined {
        if (value == undefined) {
            return undefined;
        } else {
            return func(value);
        }
    }
}
```

Итерация по таким типам невозможна, но функция map() для них вполне оправданна. Опишем еще один простой обобщенный тип, Box<T>, по сути, просто обертку для значения типа T, приведенный в листинге 11.5.

Листинг 11.5. Тип Box

```
class Box<T> {
    value: T;
    constructor(value: T) {
        this.value = value;
    }
}
```

Тип Box<T>, по сути, просто обертка для значения типа T

Можно ли отобразить такой тип данных с помощью функции `(value: T) => U`? Да. Как вы догадываетесь, функция `map()` для `Box<T>` должна возвращать `Box<U>`: она должна извлечь из `Box<T>` значение типа `T`, применить к нему функцию, а затем снова поместить результат в `Box<U>`, как показано на рис. 11.3 и в листинге 11.6.

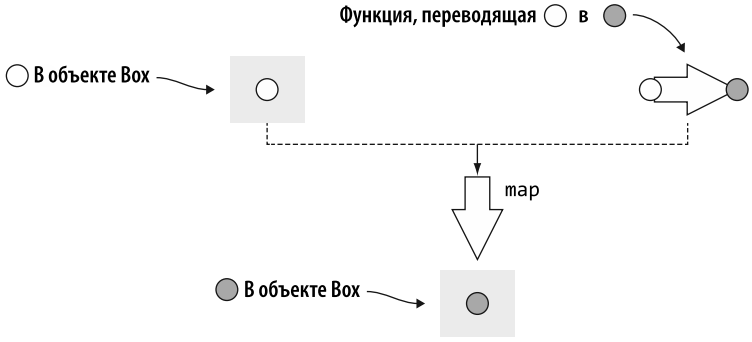


Рис. 11.3. Отображение содержащегося в Box<T> значения с помощью функции. Функция map() распаковывает содержащееся в Box<T> значение, применяет функцию, а затем снова помещает результат в Box<U>

Листинг 11.6. Алгоритм map() для типа Box

```
namespace Box {
    export function map<T, U>(
        box: Box<T>, func: (value: T) => U): Box<U> {
        return new Box<U>(func(box.value));
    }
}
```

В результате применения map() к Box<T> значение распаковывается, для него вызывается функция func(), а результат помещается в Box<U>

Доступно отображение множества типов с помощью функций. Данная возможность полезна тем, что `map()`, как и итераторы, позволяет расцепить типы, предназначенные для хранения данных, с функциями, обрабатывающими эти данные.

11.1.1. Обработка результатов и передача ошибок далее

В качестве конкретного примера рассмотрим две функции, обрабатывающие числовое значение: реализуем простую функцию `square()`, которая возвращает квадрат полученного аргумента, а также функцию `stringify()`, преобразующую свой числовой аргумент в строковое представление, как показано в листинге 11.7.

Листинг 11.7. Функции square() и stringify()

```
function square(value: number): number {
    return value ** 2;
}
function stringify(value: number): string {
    return value.toString();
}
```

Теперь пусть у нас есть функция `readNumber()`, которая читает из файла числовое значение, как показано в листинге 11.8. А поскольку речь идет о потоке ввода, не исключены потенциальные проблемы. Что, если, например, файл не существует или его не удастся открыть? В этом случае функция `readNumber()` вернет `undefined`. Мы не станем рассматривать реализацию данной функции; для нашего примера важен только ее возвращаемый тип.

Листинг 11.8. Возвращаемый тип функции readNumber()

```
function readNumber(): number | undefined {
    /* Реализация опущена */
}
```

Чтобы прочесть число и затем обработать его, применив к нему сначала функции `square()`, а затем функции `stringify()`, следует убедиться в действительном получении числового значения, а не `undefined`. Одна из возможных реализаций такой проверки — преобразование из `number | undefined` в `number` с помощью операторов `if` при необходимости, как показывает листинг 11.9.

Листинг 11.9. Обработка числа

```
function process(): string | undefined {
    let value: number | undefined = readNumber();

    if (value == undefined) return undefined;

    return stringify(square(value));
}
```

Необходимо убедиться, не undefined ли значение. В подобном случае мы сразу же возвращаем undefined

Обрабатываем значение и возвращаем результат

Наши две функции работают с числовыми значениями, однако, поскольку входные данные могут оказаться `undefined`, необходимо обрабатывать данный сценарий явным образом. Не то чтобы это плохо, но чем меньше веток в нашем коде, тем более он прост, удобен в сопровождении и понятен и тем меньше потенциальных возможностей для ошибок. Можно рассматривать это так, словно функция `process()` просто передает `undefined` дальше, ведь никаких полезных действий с ним она не производит. Лучше было бы, чтобы `process()` отвечала только за обработку данных, а какой-то другой код обрабатывал ошибки. Как же это сделать? С помощью реализованного нами для типов-сумм алгоритма `map()`, как показано в листинге 11.10.

Теперь нет никакого ветвления кода в нашей реализации функции `process()`. Обязанность распаковки `number | undefined` в `number` и проверки на `undefined` возлагается на алгоритм `map()`. Он обобщенный, его можно использовать с множеством других типов данных (например, `string | undefined`) и функций обработки.

А поскольку в нашем случае `square()` заведомо возвращает число, можно создать маленькое лямбда-выражение, которое сцепляет `square()` и `stringify()`, и передать его `map()` в листинге 11.11.

Это функциональная реализация `process()`, в которой обязанность дальнейшей передачи ошибки делегируется функции `map()`. Мы поговорим подробнее про обработку ошибок в разделе 11.2, когда будем обсуждать монады. А пока посмотрим еще на одну реализацию `map()`.

Листинг 11.10. Обработка с помощью map()

```
namespace SumType {
    export function map<T, U>(
        value: T | undefined, func: (value: T) => U): U | undefined {
        if (value == undefined) {
            return undefined;
        } else {
            return func(value);
        }
    }
}

function process(): string | undefined {
    let value: number | undefined = readNumber();

    let squaredValue: number | undefined =
        SumType.map(value, square);

    return SumType.map(squaredValue, stringify);
}
```

Это функция map() для типов-сумм, реализованная нами в листинге 11.4

Вместо того чтобы явным образом проверять на undefined, мы вызываем map() для применения к значению функции square(). Если значение undefined, то map() возвращает undefined

Аналогично функции square() вызываем map() для применения stringify() к squaredValue. Если значение undefined, то map() возвращает undefined

Листинг 11.11. Обработка с помощью лямбда-выражения

```
function process(): string | undefined {
    let value: number | undefined = readNumber();

    return SumType.map(value,
        (value: number) => stringify(square(value)));
}
```

Лямбда-выражение, передающее результат выполнения square() в stringify()

11.1.2. Сочетаем и комбинируем функции

Без семейства функций `map()` нам пришлось бы реализовывать дополнительную логику извлечения `number` из типа-суммы `number | undefined` для возводящей `number` в квадрат функции `square()`. Аналогично пришлось бы реализовывать дополнительную логику извлечения значения из `Box<number>` и упаковки его обратно в `Box<number>`, как показано в листинге 11.12.

Листинг 11.12. Распаковка значений для square()

```
function squareSumType(value: number | undefined)
    : number | undefined {
    if (value == undefined) return undefined;
    return square(value);
}

function squareBox(box: Box<number>): Box<number> {
    return new Box(square(box.value));
}
```

Функция-обертка для операции проверки на undefined

Функция распаковывает значение из Box, а затем помещает результат в еще один Box

Пока все нормально. Но что, если нам понадобится проделать то же самое с `stringify()`? Нам придется написать две функции, практически идентичные приведенным выше, как показано в листинге 11.13.

Листинг 11.13. Распаковка значений для stringify()

```
function stringifySumType(value: number | undefined)
    : string | undefined {
    if (value == undefined) return undefined;

    return stringify(value);
}

function stringifyBox(box: Box<number>): Box<string> {
    return new Box(stringify(box.value))
}
```

Начинает напоминать дублирование кода, которое ничего хорошего не сулит. Если создать функции `map()` для типов `number | undefined` и `Box`, то можно получить абстракцию, позволяющую избавиться от дублирующего кода. Можно передавать либо `square()`, либо `stringify()` в `SumType.map()` или `Box.map()`, как показано в листинге 11.14; никакого дополнительного кода не нужно.

Листинг 11.14. Использование map()

```
let x: number | undefined = 1;
let y: Box<number> = new Box(42);

console.log(SumType.map(x, stringify));
console.log(Box.map(y, stringify));

console.log(SumType.map(x, square));
console.log(Box.map(y, square));
```

Теперь опишем это семейство функций `map()`.

11.1.3. Функторы и типы, относящиеся к более высокому роду

Фактически в предыдущем подразделе мы обсуждали функторы.

ФУНКТОРЫ

> Функтор — это обобщение понятия функции для операций отображения. Для любого обобщенного типа данных, например Box<T>, операция map(), которая принимает на входе Box<T> и функцию, переводящую из T в U, и возвращает Box<U>, является функтором.

Функторы обладают исключительно широкими возможностями, но в большинстве основных языков программирования до сих пор нет удачного способа их выра-

жения, поскольку в основе общего определения функтора лежат типы, относящиеся к более высокому роду.

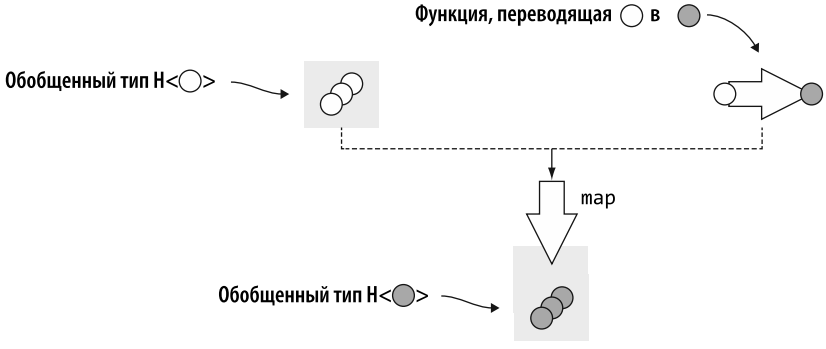


Рис. 11.4. Допустим, даны обобщенный тип данных H, содержащий ноль, одно значение или более типа T, и функция, переводящая из T в U. В данном случае T представляет собой пустой круг, а U — заполненный. Функтор map() распаковывает значение (-я) T из экземпляра H<T>, применяет функцию, после чего снова помещает полученный результат в H<U>

ТИПЫ, ОТНОСЯЩИЕСЯ К БОЛЕЕ ВЫСОКОМУ РОДУ

Обобщенный тип содержит тип-параметр. В качестве примера можно привести обобщенный тип данных T или тип наподобие Box<T> с типом-параметром T. Тип, относящийся к более высокому роду (higher kinded type), например функция высшего порядка, включает тип-параметр со своим типом-параметром. Например, типы T<U> и Box<T<U>> содержат тип-параметр T, у которого есть свой тип-параметр U.

Конструкторы типов

В системах типов *конструктором типа* (type constructor) называется функция, возвращающая тип. Самостоятельно мы их не реализуем; это часть внутреннего механизма системы типов.

У каждого типа есть конструктор. Некоторые конструкторы тривиальны. Например, конструктор для типа `number` можно рассматривать как функцию без аргументов, возвращающую тип `number`. Это будет `() -> [тип number]`.

Даже у такой функции, как `square()`, с типом `(value: number) => number`, конструктор типа все равно без аргументов `() -> [(value: number) => тип number]`, поскольку, хотя функция принимает аргумент, но ее тип не принимает тип-параметр, он всегда один и тот же.

При переходе к обобщенным типам данных ситуация приобретает еще более интересный оттенок. Для генерации конкретного типа данных из обобщенного типа наподобие `T[]` фактический тип-параметр не требуется. Его конструктор типа имеет вид `(T) -> [тип T[]]`. Например, если в качестве `T` используется `number`, то нашим типом

становится массив числовых значений `number[]`, а если роль `T` играет `string`, то получается массив строк `string[]`. Подобный конструктор также называется *«род»* — получается род типов `T[]`.

Типы, относящиеся к более высокому роду, например функции высшего порядка, — это еще более высокий уровень абстракции. В данном случае наш конструктор типа может принимать в качестве аргумента другой конструктор типа. Рассмотрим `T<U>[]` — массив значений некоего типа `T` с типом-аргументом `U`. Первый конструктор типа получает `U` и возвращает `T<U>`. Полученное необходимо передать во второй конструктор типа, генерирующий на его основе `T<U>[] ((U) -> [тип T<U>]) -> [тип T<U>[]]`.

Подобно тому как функции высшего порядка — это функции, принимающие в качестве аргументов другие функции, тип, относящийся к более высокому роду, представляет собой род (параметризованный конструктор типа), принимающий в качестве аргументов другие роды.

Теоретически количество уровней вложенности может быть произвольным (например, `T<U<V<W>>>`), однако на практике бессмысленно использовать более одного уровня (`T<U>`).

В TypeScript, C# и Java отсутствует удобный способ выражения типов, относящихся к более высокому роду, поэтому в них не получится описать с помощью системы типов конструкцию для функтора. В таких же языках, как Haskell и Idris, обладающих более развитой системой типов, подобные описания возможны. Однако в нашем случае, вследствие того что нельзя обеспечить такую возможность с помощью системы типов, она скорее может играть роль паттерна.

Можно сказать, что функтор — любой тип `H` с типом-параметром `T` (`H<T>`), для которого у нас есть функция `map()`, принимающая аргумент типа `H<T>` и функцию, переводящую из типа `T` в тип `U`, и возвращающая значение типа `H<U>`.

С другой стороны, подойдя с более объектно-ориентированной точки зрения, можно сделать `map()` функцией-членом и говорить, что `H<T>` — функтор, если он включает метод `map()`, который принимает функцию, переводящую из типа `T` в тип `U`, и возвращает значение типа `H<U>`. Чтобы посмотреть, чего именно недостает в системе типов, попробуем схематично обрисовать соответствующий интерфейс. Назовем его `Functor` и объявим в нем метод `map()` в листинге 11.15.

Листинг 11.15. Набросок интерфейса Functor

```
interface Functor<T> {
    map<U>(func: (value: T) => U): Functor<U>;
}
```

Модифицируем класс `Box<T>` в листинге 11.16 так, чтобы он реализовывал этот интерфейс.

Листинг 11.16. Класс Box, реализующий интерфейс Functor

```
class Box<T> implements Functor<T> {
    value: T;
```

```
    constructor(value: T) {
        this.value = value;
    }

    map<U>(func: (value: T) => U): Box<U> {
        return new Box(func(this.value));
    }
}
```

Этот код отлично компилируется; единственная проблема — он недостаточно конкретен. В результате вызова `map()` для объекта `Box<T>` возвращается экземпляр типа `Box<U>`. Но если речь идет об интерфейсах `Functor`, то мы видим в объявлении метода `map`, что он возвращает `Functor<U>`, а не `Box<U>`. Степени конкретизации недостаточно. Необходим способ указать в объявлении интерфейса точный возвращаемый тип метода `map()` (в данном случае `Box<U>`).

Хотелось бы иметь возможность сказать компилятору: «Данный интерфейс будет реализован типом `H` с типом-параметром `T`». В листинге 11.17 показано, как бы это объявление выглядело на языке TypeScript, если бы он поддерживал типы, относящиеся к более высокому роду. Разумеется, приведенный код не компилируется.

Листинг 11.17. Интерфейс Functor

```
interface Functor<H<T>> {
    map<U>(func: (value: T) => U): H<U>;
}

class Box<T> implements Functor<Box<T>> {
    value: T;

    constructor(value: T) {
        this.value = value;
    }

    map<U>(func: (value: T) => U): Box<U> {
        return new Box(func(this.value));
    }
}
```

А раз такой возможности нет, будем рассматривать реализации `map()` просто как паттерн применения функций к обобщенным типам данных или к каким-то упакованным значениям.

11.1.4. Функторы для функций

Обратите внимание: существуют также функторы для функций. Можно отобразить заданную функцию с произвольным числом аргументов, возвращающую значение типа `T`, с помощью функции, которая принимает `T` и возвращает `U`. В результате у нас получится функция, принимающая те же аргументы, что и исходная, и возвращающая значение типа `U`. Операция `map()` в данном случае представляет собой просто композицию функций, как показано на рис. 11.5.

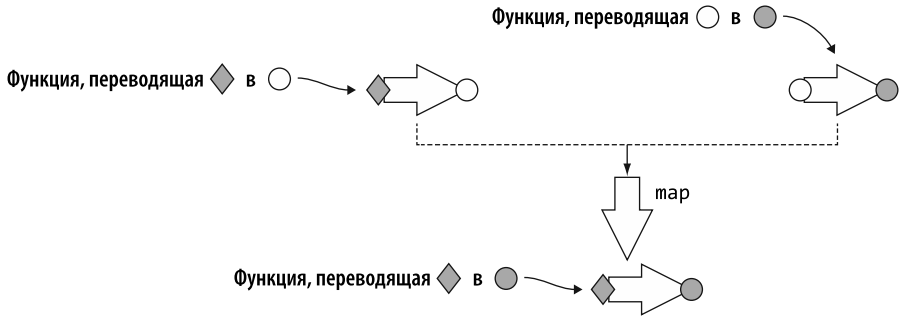


Рис. 11.5. Отображение одной функции с помощью другой означает композицию двух функций. Результат представляет собой функцию, которая принимает те же аргументы, что и первая, и выдает значение, относящееся к возвращаемому второй функцией типу. Эти две функции должны быть совместимы; вторая должна ожидать аргументы того типа, который возвращает первая

В качестве примера рассмотрим функцию, принимающую два аргумента типа `T` и выдающую значение этого же типа, и реализуем соответствующую функцию `map()` в листинге 11.18. Она возвращает функцию, которая принимает два аргумента типа `T` и возвращает значение типа `U`.

Листинг 11.18. Функция map()

```
namespace Function {
    export function map<T, U>(
        f: (arg1: T, arg2: T) => T, func: (value: T) => U)
        : (arg1: T, arg2: T) => U {
        return (arg1: T, arg2: T) => func(f(arg1, arg2));
}
}
```

map() принимает в качестве аргументов функцию типа (T, T) => T, а также вторую функцию T => U для отображения

map() возвращает функцию типа (T, T) => U

Эта реализация просто возвращает лямбда-выражение, которое выполняет композицию функций func() и f(), вызывая func() с результатом f() в качестве аргумента

Попробуем отобразить функцию `add()` (которая принимает на входе два числа и возвращает их сумму) с помощью функции `stringify()`. В результате должна получиться функция, принимающая на входе два числа и возвращающая содержащий их сумму объект `string()`, как показано в листинге 11.19.

Листинг 11.19. Применение map() для отображения функции

```
function add(x: number, y: number): number {
    return x + y;
}

function stringify(value: number): string {
    return value.toString();
}

const result: string = Function.map(add, stringify)(40, 2);
```

Функция add() просто складывает полученные аргументы

Реализация stringify() такая же, как и раньше

Отображаем функцию add() с помощью функции stringify(). И вызываем получившуюся функцию с аргументами 40 и 2. В результате получаем строку "42"

После функций нам осталось рассмотреть одну последнюю конструкцию: монады.

11.1.5. Упражнение

Допустим, дан интерфейс `IReader<T>`, в котором описан один метод: `read(): T`. Реализуйте функтор для отображения `IReader<T>` с помощью функции `(value: T) => U`, возвращающий `IReader<U>`.

11.2. Монады

Вероятно, вы уже слышали термин *«монада»* (monad), ведь в последнее время ему уделяется немало внимания. Монады понемногу появляются и в основных языках программирования, поэтому не помешает узнать о них на случай, если вы столкнетесь с ними. В данном разделе, основанном на изложенном в разделе 11.1 материале, я расскажу, что такое монады и где они могут пригодиться. Начну с нескольких примеров, а затем приведу общее определение.

11.2.1. Результат или ошибка

В разделе 11.1 упоминалась функция `readNumber()`, которая возвращала `number | undefined`. Для последовательной обработки с помощью функций `square()` и `stringify()` мы использовали функторы, так что если `readNumber()` возвращает `undefined`, то ничего не обрабатывается, просто `undefined` передается далее по конвейеру.

Подобная последовательная обработка с помощью функторов возможна в том случае, если только первая функция в цепочке — в данном случае `readNumber()` — может возвращать ошибку. Но что произойдет, если любая из сцепленных функций может выдать ошибку? Например, мы хотим открыть файл, прочитать его содержимое в строковое значение, а затем десериализовать эту строку в объект `Cat`, как показано в листинге 11.20.

Мы будем использовать функцию `openFile()`, возвращающую `Error` или `FileHandle`. Возможные причины возникновения ошибок: файл не существует, заблокирован другим процессом или у пользователя недостаточно прав для его открытия. Если же операция выполнена успешно, то функция возвращает дескриптор файла.

Кроме того, мы будем использовать функцию `readFile()`, принимающую на входе `FileHandle` и возвращающую `Error` или `string`. Возможные причины возникновения ошибок: файл не получается прочитать, например, если он слишком велик, чтобы поместиться в оперативной памяти. В случае же успешного чтения файла функция возвращает `string`.

Наконец, функция `deserializeCat()` принимает на входе строку и возвращает `Error` или экземпляр `Cat`. Возможные причины возникновения ошибок: строку

нельзя десериализовать в объект `Cat`, например, из-за отсутствия каких-либо свойств.

Все эти функции следуют паттерну «вернуть результат или ошибку» из главы 3, который предполагает возврат функцией либо допустимого результата, либо ошибки, но не обоих одновременно. Возвращаемый тип — `Either<Error, ...>`.

Листинг 11.20. Функции, возвращающие результат или ошибку

Функция readFile() возвращает Error или string

Функция openFile() возвращает Error или FileHandle

```
declare function openFile(path: string): Either<Error, FileHandle>;

declare function readFile(handle: FileHandle): Either<Error, string>;

declare function deserializeCat(
    serializedCat: string): Either<Error, Cat>;
```

Функция deserializeCat() возвращает Error или Cat

Приводить реализации я не буду, поскольку они в данном случае неважны. Теперь быстро взглянем на реализацию класса `Either` из главы 3 в листинге 11.21.

Листинг 11.21. Тип Either

Данный тип служит оберткой для значения типа TLeft или TRight, а также флага, указывающего, какой именно тип используется

```
class Either<TLeft, TRight> {
    private readonly value: TLeft | TRight;
    private readonly left: boolean;

    private constructor(value: TLeft | TRight, left: boolean) {
        this.value = value;
        this.left = left;
    }

    isLeft(): boolean {
        return this.left;
    }

    getLeft(): TLeft {
        if (!this.isLeft()) throw new Error();

        return <TLeft>this.value;
    }

    isRight(): boolean {
        return !this.left;
    }

    getRight(): TRight {
        if (!this.isRight()) throw new Error();

        return <TRight>this.value;
    }
```

Конструктор приватный, поскольку мы должны быть уверены в согласованности значения и булева флага

Попытка получения TLeft при наличии TRight или наоборот приводит к генерации ошибки

```
    static makeLeft<TLeft, TRight>(value: TLeft) {
        return new Either<TLeft, TRight>(value, true);
    }

    static makeRight<TLeft, TRight>(value: TRight) {
        return new Either<TLeft, TRight>(value, false);
    }
}
```

Функции-фабрики вызывают конструктор и обеспечивают согласованность булева флага со значением

А теперь посмотрим в листинге 11.22, как связать эти функции воедино в функцию `readCatFromFile()`, которая бы принимала в качестве аргумента путь к файлу и возвращала экземпляр `Cat` или `Error` в случае возникновения какой-либо ошибки.

Листинг 11.22. Обработка и явная проверка на ошибки

Функция readCatFromFile() возвращает Error или экземпляр Cat

Прежде всего мы пробуем открыть файл. И получаем в ответ Error или FileHandle

```
function readCatFromFile(path: string): Either<Error, Cat> {
    let handle: Either<Error, FileHandle> = openFile(path);

    if (handle.isLeft()) return Either.makeLeft(handle.getLeft());

    let content: Either<Error, string> = readFile(handle.getRight());

    if (content.isLeft()) return Either.makeLeft(content.getLeft());

    return deserializeCat(content.getRight());
}
```

Если была возвращена Error, то выполняем досрочный возврат из функции. Вызываем Either.makeLeft(), ведь нам нужно преобразовать Either<Error, FileHandle> в Either<Error, Cat>. Распаковываем Error из Either<Error, FileHandle> и снова упаковываем в Either<Error, Cat>

Наконец, после получения содержимого можно вызвать deserializeCat(). А поскольку возвращаемый тип у этой функции такой же, как и у самой readCatFromFile(), можно просто вернуть возвращенный ею результат

И снова в случае ошибки при чтении файла производим досрочный возврат из функции

Если был возвращен FileHandle, пытаемся прочитать содержимое файла

Эта функция сильно напоминает первую реализацию `process()`, приведенную ранее в данной главе. Тогда мы создали улучшенную реализацию, в которой все ветвление кода и проверка на наличие ошибок были исключены из функции и делегированы функции `map()`. Посмотрим в листинге 11.23, как могла бы выглядеть `map()` для `Either<TLeft, TRight>`. Мы следуем соглашению «правый тип соответствует корректному значению, а левый — ошибке», означающему, что `TLeft` содержит ошибку, поэтому функция `map()` будет просто передавать ее дальше. Она будет применять переданную ей функцию, только если `Either` содержит `TRight`.

Впрочем, с функцией `map()` есть одна проблема: типы ожидаемых ею в качестве аргументов функций несовместимы с нашими функциями. При такой функции `map()` после вызова `openFile()` и получения от нее `Either<Error, FileHandle>` для чтения

содержимого этого файла нам понадобится функция типа `(value: FileHandle) => string`. Данная функция не должна возвращать `Error`, но в нашем случае возможно возникновение ошибки при работе функции `readFile()`, так что она возвращает `Either<Error, string>`. Попытка воспользоваться ею в нашей `readCatFromFile()` приведет к ошибке компиляции, как показывает листинг 11.24.

Листинг 11.23. Функция map() для Either

Функция func() применяется только в том случае, если Either содержит значение типа TRight; так что тип ее аргумента должен быть TRight

Если входные данные содержат значение типа TLeft, то мы распаковываем его и заново упаковываем в Either<TLeft, URight>

```
namespace Either {
    export function map<TLeft, TRight, URight>(
        value: Either<TLeft, TRight>,
        func: (value: TRight) => URight): Either<TLeft, URight> {
        if (value.isLeft()) return Either.makeLeft(value.getLeft());

        return Either.makeRight(func(value.getRight()));
    }
}
```

Если входные данные содержат значение типа TRight, то мы распаковываем его, применяем к нему функцию func() и упаковываем полученный результат в Either<TLeft, URight>

Листинг 11.24. Несовместимость типов данных

```
function readCatFromFile(path: string): Either<Error, Cat> {
    let handle: Either<Error, FileHandle> = openFile(path);

    let content: Either<Error, string> = Either.map(handle, readFile);

    /* ... */
}
```

Из-за несоответствия типов возникает ошибка компиляции

Мы получим при этом сообщение об ошибке: `Type 'Either<Error, Either<Error, string>>' is not assignable to type 'Either<Error, string>'` (Невозможно присвоить значение типа `'Either<Error, Either<Error, string>>'` переменной типа `'Either<Error, string>'`).

Функтор не оправдал наших надежд. Функторы могут передавать возникающую на начальной этапе ошибку далее по конвейеру обработки, но если ошибка может возникать на любом из шагов конвейера, то функторы не подходят. На рис. 11.6 черный квадрат соответствует `Error`, а черный и белый круги — двум типам, например `FileHandle` и `string`.

Для отображения `Either<Error, FileHandle>` на `Either<Error, string>` необходима функция `map()`, переводящая `FileHandle` в `string`. Однако наша функция `readFile()` переводит `FileHandle` в `Either<Error, string>`.

Решить эту проблему несложно. Нужна функция, аналогичная `map()`, которая бы переводила `T` в `Either<Error, U>`, как показано в листинге 11.25. Традиционно для такой функции используется название `bind()`.

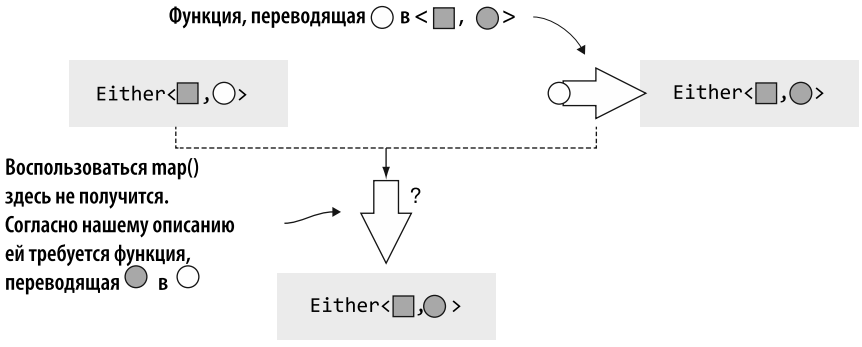


Рис. 11.6. Использовать функтор в данном случае нельзя, поскольку он определяется как операция отображения белого круга в черный с помощью заданной функции. К сожалению, наша функция возвращает уже обернутый в Either тип (Either<черный квадрат, черный круг>). Нам нужна замена map(), способная работать с подобными функциями

Листинг 11.25. Функция bind() для Either

```
namespace Either {
    export function bind<TLeft, TRight, URight>(
        value: Either<TLeft, TRight>,
        func: (value: TRight) => Either<TLeft, URight>
        ): Either<TLeft, URight> {
        if (value.isLeft()) return Either.makeLeft(value.getLeft());

        return func(value.getRight());
    }
}
```

Тип func() здесь отличается от типа func() в map()

Можно просто вернуть результат функции func(), поскольку его тип совпадает с возвращаемым типом bind()

Как можно видеть, эта реализация даже проще реализации `map()`: после распаковки значения мы просто возвращаем результат применения к нему функции `func()`. Воспользуемся функцией `bind()` для реализации нашей функции `readCatFromFile()` в листинге 11.26 и достижения требуемой передачи ошибки далее без ветвления кода.

Листинг 11.26. Реализация readCatFromFile() без ветвления кода

```
function readCatFromFile(path: string): Either<Error, Cat> {
    let handle: Either<Error, FileHandle> = openFile(path)

    let content: Either<Error, string> =
        Either.bind(handle, readFile);

    return Either.bind(content, deserializeCat);
}
```

В отличие от варианта с map(), этот код вполне работоспособен. В результате применения функции readFile() к дескриптору возвращается Either<Error, string>

Возвращаемый тип функции deserializeCat() совпадает с возвращаемым типом функции readCatFromFile(), так что мы просто возвращаем результат выполнения функции bind()

Эта версия органично сцепляет функции `openFile()`, `readFile()` и `deserializeCat()` так, что в случае сбоя любой из них ошибка передается далее в качестве результата `readCatFromFile()`. Опять же все ветвление кода инкапсулируется в реализации `bind()`, и наша функция-обработчик оказывается линейной.

11.2.2. Различия между map() и bind()

Прежде чем перейти к определению монады, рассмотрим еще один упрощенный пример и сравним `map()` и `bind()`. Мы снова воспользуемся `Box<T>` — обобщенным типом данных — оберткой для типа `T`. Он не особенно полезен, однако это простейший обобщенный тип данных, а мы хотели бы сосредоточиться на работе функций `map()` и `bind()` со значениями типов `T` и `U` в контексте обобщенных типов данных, таких как `Box<T>`, `Box<U>` (или `T[]`, `U[]`; или `Optional<T>`, `Optional<U>`; или `Either<Error, T>`, `Either<Error, U>` и т. д.).

В случае `Box<T>` функтор (`map()`) принимает на входе `Box<T>` и функцию, переводящую `T` в `U`, и возвращает `Box<U>`. Проблема в том, что в некоторых сценариях наши функции переводят `T` непосредственно в `Box<U>`. Именно в них нам и пригодится функция `bind()`. Она принимает на входе `Box<T>` и функцию, переводящую `T` в `Box<U>`, и возвращает результат применения этой функции к значению `T` внутри `Box<T>` (рис. 11.7).

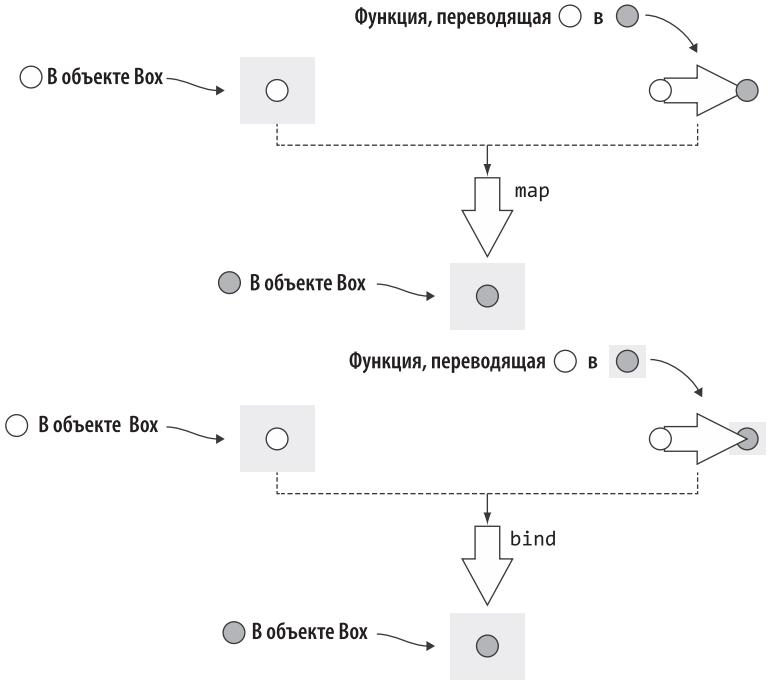


Рис. 11.7. Сравнение map() и bind(). Функтор map() применяет к значению Box<T> функцию T => U и возвращает Box<U>. Функция bind() применяет к значению Box<T> функцию T => Box<U> и возвращает Box<U>

С помощью функции `stringify()`, которая принимает в качестве аргумента число и возвращает его строковое представление, можно отобразить `Box<number>` и получить `Box<string>`, как показано в листинге 11.27.

Листинг 11.27. Применение map() к Box

```
namespace Box {
    export function map<T, U>(
        box: Box<T>, func: (value: T) => U): Box<U> {
        return new Box<U>(func(box.value));
    }
}

function stringify(value: number): string {
    return value.toString();
}

const s: Box<string> = Box.map(new Box(42), stringify);
```

Приведенная ранее в этой главе реализация функции map() для Box

Приведенная ранее в этой главе реализация функции stringify(), которая принимает в качестве аргумента число и возвращает строку

Отображаем Box<number> с помощью stringify() и получаем Box<string>

Но для функции `boxify()`, переводящей `number` сразу в `Box<string>` (вместо `stringify()`, переводящей `number` в `string`), `map()` не подходит. Вместо нее понадобится `bind()`, как показано в листинге 11.28.

Листинг 11.28. Применение bind() к Box

```
namespace Box {
    export function bind<T, U>(
        box: Box<T>, func: (value: T) => Box<U>): Box<U> {
        return func(box.value);
    }
}

function boxify(value: number): Box<string> {
    return new Box(value.toString());
}

const b: Box<string> = Box.bind(new Box(42), boxify);
```

bind() распаковывает значение из Box и вызывает для него func()

Функция boxify() отличается от stringify() тем, что возвращает Box<string> вместо просто string

Можно воспользоваться bind для boxify() и Box<number> и получить в качестве результата Box<string>

Результат обеих функций — и `map()`, и `bind()` — `Box<string>`. В обоих случаях мы переводим `Box<T>` в `Box<U>`; разница в том, как мы этого добиваемся. Применительно к функции `map()` необходима функция, переводящая `T` в `U`; в случае же функции `bind()` необходима функция, переводящая `T` в `Box<U>`.

11.2.3. Паттерн «Монада»

Монада состоит из `bind()` и еще одной, более простой функции. Эта другая функция принимает аргумент типа `T` и обертывает его в обобщенный тип данных, например `Box<T>`, `T[]`, `Optional<T>`, `Either<Error, T>`. Обычно эта функция называется `return()` или `unit()`.

С помощью монад можно структурировать программы обобщенно, инкапсулируя одновременно нужный для логики программы стереотипный код. Монады позволяют представлять последовательность вызовов функций в виде конвейера, абстрагирующего управление данными, поток команд и побочные эффекты.

Рассмотрим несколько примеров монад. Начнем с нашего простого типа `Box<T>`, дополнив его до монады функцией `unit()` в листинге 11.29.

Листинг 11.29. Монада Box

```
namespace Box {
    export function unit<T>(value: T): Box<T> {
        return new Box(value);
    }

    export function bind<T, U>(
        box: Box<T>, func: (value: T) => Box<U>): Box<U> {
        return func(box.value);
    }
}
```

Функция unit() просто вызывает конструктор Box для обертывания заданного значения в экземпляр Box<T>

bind() распаковывает значение из Box и вызывает для него func()

Данная реализация очень проста. Посмотрим на функции монады `Optional<T>` в листинге 11.30.

Листинг 11.30. Монада Optional

```
namespace Optional {
    export function unit<T>(value: T): Optional<T> {
        return new Optional(value);
    }

    export function bind<T, U>(
        optional: Optional<T>,
        func: (value: T) => Optional<U>): Optional<U> {
        if (!optional.hasValue()) return new Optional();

        return func(optional.getValue());
    }
}
```

Функция unit() принимает в качестве аргумента значение типа T и обертывает его в Optional<T>

Если опционал пуст, то bind() возвращает пустой опционал типа Optional<U>

Если опционал содержит значение, то bind() возвращает результат применения к нему функции func()

Как и в случае функторов, не существует хорошего способа задать интерфейс `Monad`, если язык программирования не способен выражать типы, относящиеся к более высокому роду. В этом случае можно рассматривать монады как паттерн.

ПАТТЕРН «МОНАДА»

Монада (monad) — это обобщенный тип данных H<T>, для которого существует функтор наподобие unit(), принимающий значение типа T и возвращающий значение типа H<T>, а также функция bind(), которая принимает в качестве аргументов значение типа H<T> и функцию, переводящую из T в H<U>, и возвращает значение типа H<U>.

Помните: большинство языков программирования использует данный паттерн, не предоставляя какого-либо способа задать интерфейс, соответствие которому мог бы проверить компилятор. Как следствие, эти две функции, `unit()` и `bind()`, встречаются под различными названиями. Возможно, вам уже встречался термин *«монадический»* (monadic), например, в словосочетании *«монадическая обработка ошибок»* (monadic error handling), означающем, что способ обработки ошибок следует паттерну «Монада».

Далее мы рассмотрим еще один пример. Возможно, вы удивитесь, поскольку он уже встречался ранее, в главе 6; просто тогда мы еще не знали, как это называется.

11.2.4. Монада продолжения

В главе 6 мы изучали способы упрощения асинхронного кода и закончили на промисах. *Промис* отражает результат вычисления, которое будет выполнено когда-либо в будущем. `Promise<T>` — промис для значения типа `T`. Можно планировать выполнение асинхронного кода путем сцепления промисов с помощью функции `then()`.

Допустим, у нас есть функция определения нашего местоположения на карте. Выполнение этой функции может занять длительное время, поскольку она использует GPS, так что сделаем ее асинхронной. Она будет возвращать промис типа `Promise<Location>`. Далее представим, что у нас есть функция, которая получает местоположение и связывается с сервисом совместных поездок, чтобы арендовать для нас машину (`Car`), как демонстрирует листинг 11.31.

Листинг 11.31. Сцепление промисов

```
declare function getLocation(): Promise<Location>;
declare function hailRideshare(location: Location): Promise<Car>;

let car: Promise<Car> = getLocation().then(hailRideshare);
```

После возврата значения функцией getLocation() для ее результата вызывается функция hailRideshare()

Возможно, это выглядит для вас очень знакомо; `then()`, по существу, просто версия `bind()` для `Promise<T>`!

Как мы видели в главе 6, можно также создать промис, который разрешается сразу же, с помощью `Promise.resolve()`. Данный метод принимает значение и возвращает содержащий это значение уже разрешенный промис. Это эквивалент функции `unit()` для `Promise<T>`.

Оказывается, доступное практически во всех основных языках программирования API сцепление промисов — монадическое. Оно следует паттерну, который мы видели ранее в этом подразделе, только для другой предметной области. При обработке ошибок в монаде инкапсулируется проверка полученного нами значения для дальнейшей обработки или ошибки, которую необходимо передать далее. В случае промисов монада инкапсулирует нюансы планирования и возобновления выполнения. Паттерн, впрочем, остается тем же самым.

11.2.5. Монада списка

Часто используется и монада списка. Рассмотрим реализацию для последовательностей: функцию `divisors()`, которая возвращает по заданному числу `n` массив, включающий все его делители, за исключением 1 и самого `n`, как показано в листинге 11.32.

Эта бесхитростная реализация проходит, начиная с 2 и до половины числа `n`, собирая все найденные числа, на которые `n` делится без остатка. Существуют намного более эффективные способы поиска всех делителей числа, но для примера нам достаточно этого простого алгоритма.

Листинг 11.32. Делители числа

```
function divisors(n: number): number[] {
    let result: number[] = [];

    for (let i = 2; i <= n / 2; i++) {
        if (n % i == 0) {
            result.push(i);
        }
    }

    return result;
}
```

Допустим теперь, что, получив в качестве аргумента массив чисел, мы хотим вернуть массив, содержащий все делители всех чисел из него. Дублирование нас не волнует. Одна из возможных реализаций: написать функцию, которая принимает в качестве аргумента входной массив чисел, применяет к каждому из них функцию `divisors()` и объединяет результаты всех этих вызовов `divisors()` в итоговый результат, как показано в листинге 11.33.

Листинг 11.33. Все делители чисел

```
function allDivisors(ns: number[]): number[] {
    let result: number[] = [];

    for (const n of ns) {
        result = result.concat(divisors(n));
    }

    return result;
}
```

Оказывается, этот паттерн встречается очень часто. Допустим, у нас есть еще одна функция, `anagram()`, которая генерирует список всех перестановок символов строки и возвращает массив строк. Чтобы получить набор всех анаграмм массива строк, нам придется реализовать очень похожую функцию, как показывает листинг 11.34.

Листинг 11.34. Все анаграммы

```
declare function anagram(input: string): string[];

function allAnagrams(inputs: string[]): string[] {
    let result: string[] = [];

    for (const input of inputs) {
        result = result.concat(anagram(input));
}

    return result;
}
```

Реализация функции anagram() опущена

Функция allAnagrams() очень напоминает функцию allDivisors()

Попробуем теперь заменить функции `allDivisors()` и `allAnagrams()` одной обобщенной функцией в листинге 11.35. Эта функция должна принимать массив значений типа `T` и функцию, переводящую `T` в массив `U`, и возвращать массив значений типа `U`.

Листинг 11.35. Функция bind() для списка

```
function bind<T, U>(inputs: T[], func: (value: T) => U[]): U[] {
    let result: U[] = [];

    for (const input of inputs) {
        result = result.concat(func(input));
    }

    return result;
}

function allDivisors(ns: number[]): number[] {
    return bind(ns, divisors);
}

function allAnagrams(inputs: string[]): string[] {
    return bind(inputs, anagram);
}
```

bind() принимает массив значений типа T и функцию, которая по данному T возвращает массив U, и возвращает массив значений типа U

Применяем функцию func() к каждому полученному на входе значению типа T и склеиваем результаты

Функцию allDivisors() можно реализовать, применив bind() к массиву чисел и функции divisors()

Функцию allDivisors() можно реализовать, применив bind() к массиву строк и функции anagram()

Как вы, возможно, догадались, это реализация функции `bind()` для монады списка. Относительно списков эта функция схлопывает массивы, возвращаемые в результате отдельных вызовов заданной функции, в единый массив. Монада распространения ошибки определяет, передавать ли ошибку далее или применить функцию, а монада продолжения служит оберткой для планирования выполнения. А вот монада списка объединяет набор результатов (список списков) в единый схлопнутый список. В данном случае аналогом `Box` служит последовательность значений (рис. 11.8).

Реализация функции `unit()` элементарна. По заданному значению типа `T` она возвращает список, содержащий одно это значение. Данная монада обобщается на все разновидности списков: массивы, связные списки и интервалы итераторов.

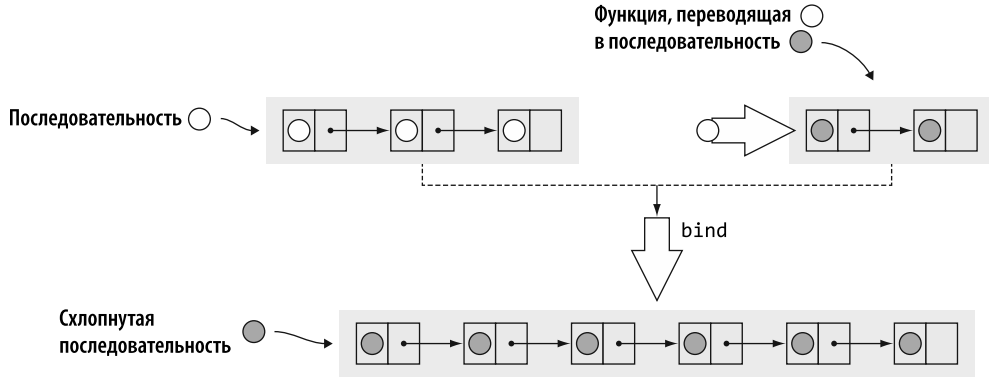


Рис. 11.8. Монада списка: bind() принимает последовательность значений типа T (в данном случае белых кругов) и функцию типа «значение типа T => последовательность значений типа U» (в данном случае черных кругов). Результат представляет собой схлопнутый список значений типа U (черных кругов)

Теория категорий

Функторы и монады — понятия из теории категорий, раздела математики, посвященного структурам, состоящим из объектов, и отношениям между ними. Из таких маленьких стандартных блоков можно формировать структуры наподобие функторов и монад. Мы не станем обсуждать подробности, а просто усвоим, что на языке этой теории можно описывать понятия из различных областей, например теории множеств и даже систем типов.

Язык программирования Haskell был в немалой степени вдохновлен теорией категорий, поэтому его синтаксис и стандартная библиотека позволяют с легкостью выражать такие понятия, как функторы, монады и другие структуры. Haskell в полной мере поддерживает типы, относящиеся к более высокому роду.

Возможно, именно вследствие простоты стандартных блоков теории категорий обсуждавшиеся нами абстракции применимы в таком широком спектре областей. Как мы только что видели, монады удобны в контексте распространения ошибок, асинхронного кода и обработки последовательностей.

Хотя в большинстве основных языков программирования монады все еще рассматриваются как паттерны, а не настоящие конструкции языка, это, безусловно, удобные структуры, повсеместно встречающиеся в различных контекстах.

11.2.6. Прочие разновидности монад

Два других распространенных вида монад, часто встречающихся в функциональных языках программирования с чистыми функциями (функциями без побочных эффектов) и неизменяемыми данными, — монада состояния и монада ввода/вывода. Здесь мы рассмотрим их весьма поверхностно, но если вы решите изучать функцио-

нальный язык программирования, такой как Haskell, то наверняка очень быстро столкнетесь с ними.

Монада состояния инкапсулирует часть состояния для передачи вместе со значением. Благодаря этой монаде можно создавать чистые функции, возвращающие на основе текущего состояния значение и обновленное состояние. Сцепление их с `bind()` позволяет передавать дальше по конвейеру и обновлять состояние, не прибегая к сохранению его в переменной явным образом, благодаря чему чисто функциональный код оказывается способен обрабатывать и модифицировать состояние.

Монада ввода/вывода инкапсулирует побочные эффекты. Она дает возможность реализовывать чистые функции, способные читать вводимые пользователем данные или производить запись в файл или вывод в терминал благодаря выносу из функции не являющегося чистым поведения и обертывания его в монаду.

В разделе 11.3 приведены источники, из которых можно почерпнуть дополнительную информацию о монадах, если эта тема вас заинтересовала.

11.2.7. Упражнение

Рассмотрим функциональный тип данных `Lazy<T>` вида `() => T`, то есть функцию без аргументов, возвращающую значение типа `T`. Он выполняется отложенным образом, поскольку возвращает значение типа `T`, но делает это только по запросу. Реализуйте функции `unit()`, `map()` и `bind()` для данного типа.

11.3. Что изучать дальше

В данной книге мы обсудили множество тем, начиная от простых типов данных и композиции до функциональных типов данных, подтипизации, обобщенных типов данных и фрагмента систем типов. В этом последнем разделе мы обсудим еще несколько тем, которые, возможно, вам захочется изучить более детально, и узнаем, с чего начинать изучение каждой из них.

11.3.1. Функциональное программирование

Парадигма функционального программирования очень сильно отличается от объектно-ориентированного программирования. Изучение языка функционального программирования позволит вам взглянуть на написание кода совершенно с другой стороны. А чем больше путей решения задачи вам известно, тем легче ее проанализировать и решить.

Нефункциональные языки программирования включают все больше возможностей и паттернов функционального программирования, что свидетельствует об их широкой применимости. Лямбда-выражения и замыкания, неизменяемые структуры данных и реактивное программирование — все они пришли из функционального программирования.

Лучший способ начать знакомство со всем этим — выбрать для изучения какой-либо функциональный язык программирования. Я рекомендую начать с Haskell.

Он отличается очень простым синтаксисом и мощной системой типов, а также прочным теоретическим фундаментом. Отличное легкочитаемое вводное руководство по нему — книга Мирана Липовачи *Learn You a Haskell for Great Good!*[1], опубликованная издательством No Starch Press.

11.3.2. Обобщенное программирование

Как мы видели в предыдущих главах, обобщенное программирование — ключ к совершенно замечательным абстракциям и повторному использованию кода. Популярность этого стиля программирования началась со стандартной библиотеки шаблонов C++ и его набора комбинируемых структур данных и алгоритмов.

Истоки обобщенного программирования лежат в абстрактной алгебре. Александр Степанов, сочинивший термин *«обобщенное программирование»* и реализовавший первую библиотеку шаблонов, написал две посвященные этой теме книги: *Elements of Programming* (в соавторстве с Полом Мак-Джонсом) и *From Mathematics to Generic Programming*[2] (в соавторстве с Дэниелом И. Роузом), опубликованные издательством Addison-Wesley Professional.

Обе книги насыщены теоретической математикой, но, я надеюсь, это вам не помешает. Элегантность и красота кода потрясают. Основная их идея: при наличии нужных абстракций в компромиссах нет нужды: код может быть одновременно лаконичным, производительным, читабельным и элегантным.

11.3.3. Типы, относящиеся к более высокому роду, и теория категорий

Ранее я уже упоминал, что такие конструкции, как функторы, пришли непосредственно из теории категорий. Введением в эту тему, написанным чрезвычайно простым языком, может послужить книга *Category Theory for Programmers*[3] Бартоша Милевски (самиздат).

Мы обсудили функторы и монады, но это далеко не все типы, относящиеся к более высокому роду. Вероятно, пройдет немало времени, прежде чем данные возможности появятся в более распространенных языках программирования, но если вы хотели бы забежать немного наперед, то для изучения этих понятий отлично подойдет язык Haskell.

Возможность задавать такие высокоуровневые абстракции, как монады, позволяет писать код, который еще удобнее переиспользовать.

---

[1] *Миран Л.* Изучай Haskell во имя добра! Для начинающих. — М.: ДМК Пресс, 2017.

[2] *Степанов А., Мак-Джонс П.* Начала программирования. — М.: Вильямс, 2011. *Роуз Д., Степанов А. А.* От математики к обобщенному программированию. — М.: ДМК Пресс, 2015.

[3] Теория категорий для программистов. Неофициальный перевод можно найти на сайте https://henrychern.wordpress.com/2017/07/17/httpsbartoszmilewski-com20141028category-theory-for-programmers-the-preface/. — *Примеч. пер.*

11.3.4. Зависимые типы данных

В этой книге недостаточно места для обсуждения зависимых типов данных, но если вы хотите узнать больше о том, как хорошая система типов способна обеспечить безопасность кода, то данная тема вас заинтересует.

Если очень коротко: мы уже видели, что тип способен указывать, какие значения может принимать переменная. Мы также рассмотрели обобщенные типы данных, в которых тип может указывать, каким должен быть другой тип данных (типы-параметры). Зависимые типы — нечто обратное: значения указывают, каким должен быть тип. Классический пример: кодирование длины списка в системе типов. Тип числового списка из двух элементов при этом отличается от типа числового списка из пяти элементов. А их конкатенация дает третий тип: список из семи элементов. Возможно, вы уже понимаете, как благодаря кодированию подобной информации в системе типов можно гарантировать, что индекс никогда не выйдет за допустимые пределы.

Если вы хотели бы узнать больше о зависимых типах, то я рекомендую книгу *Type Driven Development with Idris* Эдвина Брэди издательства Manning. Idris — язык программирования, синтаксис которого очень напоминает Haskell, но включена поддержка зависимых типов данных.

11.3.5. Линейные типы данных

В главе 1 мы мельком упомянули глубинную связь между системами типов и логикой. Линейная логика — отличный от классической логики подход, ориентированный на работу с ресурсами. В классической логике истинное умозаключение остается истинным всегда, а вот в доказательстве в линейной логике умозаключения могут использоваться только один раз.

У линейной логики есть непосредственное приложение в языках программирования, в которых с ее помощью в системе типов кодируется отслеживание использования ресурсов. Rust — один из языков программирования, чья популярность непрерывно растет; в нем линейные типы служат для обеспечения безопасности ресурсов. Встроенное средство проверки заимствований (borrow checker) языка Rust гарантирует, что у любого ресурса только один владелец. При передаче объекта в функцию происходит смена владельца ресурса, и компилятор больше не позволяет ссылаться на данный ресурс, пока функция его не вернет. Это делается для устранения проблем конкурентности, а также столь опасных ситуаций использования ресурса после освобождения памяти и повторного освобождения памяти, нередко встречающихся в языке C.

Изучить Rust имеет смысл хотя бы за его расширенную поддержку обобщенных типов данных и уникальные возможности обеспечения безопасности. На сайте Rust можно бесплатно скачать книгу *The Rust Programming Language* Стива Клабника и Кэрол Николс при участии сообщества Rust — прекрасное введение в этот язык (https://doc.rust-lang.org/book).

Резюме

- Функцию `map()` можно обобщить с итераторов и на другие типы данных.
- Функторы инкапсулируют распаковку данных и могут применяться при композиции и для передачи ошибок далее.
- Типы, относящиеся к более высокому роду, позволяют выражать подобные функторам конструкции путем применения обобщенных типов данных, у которых есть свои типы-параметры.
- С помощью монады распространения ошибок можно сцеплять операции, возвращающие результат или ошибку, инкапсулируя таким образом логику передачи ошибки далее.
- Промисы — это монады, инкапсулирующие планирование выполнения/асинхронное выполнение кода.
- Монада списка применяет функцию генерации последовательности к последовательности значений и возвращает схлопнутую последовательность.
- В языках программирования, которые не поддерживают типы, относящиеся к более высокому роду, функторы и монады можно рассматривать как паттерны, применимые для решения разнообразных задач.
- Haskell — язык, изучение которого может помочь в освоении функционального программирования и типов, относящихся к более высокому роду.
- Idris — язык, изучение которого может помочь в освоении зависимых типов данных и их приложений.
- Rust — язык, изучение которого может помочь в освоении линейных типов данных и их приложений.

Надеюсь, эта книга вам понравилась, вы почерпнули из нее то, что пригодится в вашей работе, и благодаря ей смогли взглянуть на некоторые вещи с новой точки зрения. Удачи вам в типобезопасном программировании!

Ответы к упражнениям

11.1. Еще более обобщенная версия алгоритма map

Одна из возможных реализаций включает использование объектно-ориентированного паттерна «Декоратор», к которому мы обращались в главе 5. Таким образом можно создать тип, реализующий интерфейс `IReader<U>`, который служил бы оберткой для `IReader<T>` и при вызове метода `read()` отображал бы исходное значение с помощью заданной функции:

```
interface IReader<T> {
    read(): T;
}
```

```
namespace IReader {
    class MappedReader<T, U> implements IReader<U> {
        reader: IReader<T>;
        func: (value: T) => U;

        constructor(reader: IReader<T>, func: (value: T) => U) {
            this.reader = reader;
            this.func = func;
        }

        read(): U {
            return this.func(this.reader.read());
        }
    }

    export function map<T, U>(reader: IReader<T>, func: (value: T) => U)
        : IReader<U> {
        return new MappedReader(reader, func);
    }
}
```

11.2. Монады

Ниже приведена одна из возможных реализаций. Обратите внимание на различие между `map()` и `bind()`.

```
type Lazy<T> = () => T;

namespace Lazy {
    export function unit<T>(value: T): Lazy<T> {
        return () => value;
    }

    export function map<T, U>(lazy: Lazy<T>, func: (value: T) => U)
        : Lazy<U> {
        return () => func(lazy());
    }

    export function bind<T, U>(lazy: Lazy<T>, func: (value: T) => Lazy<U>)
    : Lazy<U> {
        return func(lazy());
    }
}
```

Приложение А Установка TypeScript и исходный код

Онлайн

Для простого кода, например, чтобы попробовать в действии некоторые из примеров кода без зависимостей, вы можете воспользоваться онлайн-песочницей TypeScript по адресу https://www.typescriptlang.org/play.

На локальной машине

Для установки TypeScript на локальной машине необходимо сначала установить Node.js и систему управления пакетами Node — npm. Скачать их можно по адресу https://www.npmjs.com/get-npm. После этого выполните команду `npm install -g typescript` для установки компилятора TypeScript.

Скомпилировать отдельный файл TypeScript можно, передав его в качестве аргумента компилятору TypeScript, вот так: `tsc helloworld.ts`. TypeScript компилирует в JavaScript.

Для содержащих несколько файлов проектов используется файл `tsconfig.json`, позволяющий задать настройки компилятора. Для компиляции всего проекта в соответствии с этой конфигурацией достаточно выполнить команду `tsc` без аргументов в содержащем файл `tsconfig.json` каталоге.

Исходный код

Примеры кода из этой книги можно найти по адресу https://github.com/vladris/programming-with-types. Код для каждой главы размещен в отдельном каталоге со своим файлом `tsconfig.json`.

Сборка кода производилась с помощью версии 3.3 компилятора TypeScript со значением ES6 для опции `target` и опцией `strict`.

Все файлы примеров самодостаточны, в каждый из них встроены все типы и функции, необходимые для запуска примера кода. Во всех файлах примеров используются уникальные пространства имен во избежание конфликтов имен, поскольку в некоторых примерах приведены различные реализации одних и тех же функций или паттернов.

Для запуска файла примера сначала скомпилируйте его с помощью `tsc`, а затем запустите скомпилированный JavaScript-файл, используя Node. Например, после компиляции, инициированной командой `tsc helloworld.ts`, можно запустить код, задействовав команду `node helloworld.js`.

«Самодельные» реализации

В данной книге описываются «самодельные» реализации типа variant и других типов данных в TypeScript. Версии C# и Java этих типов можно найти в библиотеке типов Maki: https://github.com/vladris/maki.

Приложение Б
Шпаргалка по TypeScript

Эта шпаргалка отнюдь не исчерпывающая. Она охватывает лишь ту часть синтаксиса TypeScript, которая применяется в данной книге. Полный справочник по TypeScript можно найти на сайте http://www.typescriptlang.org/docs.

Таблица Б.1. Простые типы данных

Тип	Описание
`boolean`	Может принимать значение true или false
`number`	64-битное число с плавающей точкой
`string`	Строка в кодировке Unicode UTF-16
`void`	Используется в качестве возвращаемого типа для функций, не возвращающих никакого осмысленного значения
`undefined`	Может принимать только значение undefined. Может представлять, например, объявленную, но не получившую начального значения переменную
`null`	Может принимать только значение null
`object`	Представляет object (тип данных, не являющийся простым)
`unknown`	Может представлять любое значение. Типобезопасный, поскольку не преобразуется автоматически в другие типы
`Any`	Обходит проверку типов. Не обеспечивает типобезопасность и автоматически преобразуется в любой другой тип
`Never`	Не может отражать никакого значения

Таблица Б.2. Составные типы данных

Пример	Описание
`string[]`	Типы массивов обозначаются с помощью указания [] после названия типа — в данном случае это массив строк
`[number, string]`	Кортежи объявляются с помощью перечисления типов в [] — в данном случае типов number и string, например [0, "hello"]

Пример	Описание
`(x: number, y: number) => number;`	Функциональные типы данных объявляются в виде списка аргументов, следующего за ними символа =>, а затем возвращаемого типа данных

`enum Direction {`
`  North,`
`  East,`
`  South,`
`  West,`
`}`
	Перечисляемые типы данных объявляются с помощью ключевого слова enum. В данном случае значение может быть одним из North, East, South и West

`type Point {`
`  X: number,`
`  Y: number`
`}`
	Тип, включающий свойства X и Y типа number

`interface IExpression {`
`  evaluate(): number;`
`}`
	Интерфейс с методом evaluate(), возвращающим number

`class Circle extends Shape`
`  implements IGeometry {`
`  // ...`
`}`
	Класс Circle расширяет базовый класс Shape и реализует интерфейс IGeometry

`type Shape = Circle | Square;`	Типы-объединения объявляются в виде списка типов, разделенного символом |. Shape может быть либо Circle, либо Square
`type SerializableExpression =` `  = Serializable & Expression;`	Типы-пересечения объявляются в виде списка типов, разделенного символом &. SerializableExpression включает все члены типа Serializable и все члены типа Expression

Таблица Б.3. Объявления

Пример	Описание
`let x: number = 0;`	Объявление переменной x типа number с начальным значением 0
`let x: number;`	Объявление переменной x типа number, которой перед использованием необходимо присвоить значение
`const x: number = 0;`	Объявление константы x типа number со значением 0. Изменение значения x невозможно

`function add(x: number, y: number)`
`: number {`
`return x + y;`
`}`
	Объявление функции add(), получающей два аргумента, x и y типа number, и возвращающей number

`(x: number, y: number) => x + y;`	Лямбда-выражение (анонимная функция), принимающее два аргумента и возвращающее их сумму

Продолжение ⇨

Таблица Б.3 *(продолжение)*

Пример	Описание

`namespace Ns {`
`  export function func(): void {`
`  }`
`}`
`Ns.func();`
	Пространства имен объявляются с помощью ключевого слова namespace. Чтобы объявления внутри пространства имен были видимы извне, необходимо перед ними указывать export

`class Example {`
`  a: number = 0;`
`  private b: number = 0;`
`  protected c: number = 0;`
`  readonly d: number;`
 	
`  constructor(d: number) {`
`  this.d = d;`
`  }`
`  getD(): number {`
`  return this.d;`
`  }`
`}`
 	
`let instance: Example`
`  = new Example(5);`
	По умолчанию все члены класса публичны (public). Они также могут быть защищенными (protected), то есть видимыми только членам производных классов, и приватными (private) — видимыми только внутри самого класса. Свойства также могут быть readonly, и в таком случае их нельзя модифицировать после установки начального значения. И если для свойства не разрешено undefined в качестве значения, то оно должно быть инициализировано либо на месте, либо с помощью конструктора. Конструктор для любого класса называется constructor(). Перед ссылками на члены класса внутри него необходимо указывать this. Для создания объектов используется ключевое слово new, вызывающее конструктор

`declare const Sym: unique symbol;`	Компилятор гарантирует уникальность Sym. Равенство никаких двух констант, объявленных как unique symbol, невозможно

Таблица Б.4. Обобщенные типы данных

Объявление	Описание

`function identity<T>(value: T): T {`
`  return value;`
`}`
	У обобщенной функции указывается один или несколько типов-параметров внутри <> перед списком аргументов. У этой функции identity() один тип-аргумент T. Она принимает значение типа T и возвращает также значение типа T

`let str: string =` `  identity<string>("Hello");`	При указании конкретного типа внутри <> создается конкретный экземпляр обобщенной функции. identity<string>() представляет собой функцию identity(), в которой роль типа T играет string

`class Box<T> {`
`  value: T;`
 	
`  constructor(value: T) {`
`  this.value = value;`
`  }`
`}`
 	
`let x: Box<number> = new Box(0);`
	У обобщенных классов указывается один или несколько типов-параметров между <> после названия класса. Класс Box включает значение-свойство типа T.
	При указании конкретного типа внутри <> создается конкретный экземпляр обобщенного класса. Box<number> представляет собой класс Box, в котором роль T играет number

`class Expr<T extends IExpression> {`
`  /* ... */`
`}`
	Ограничение обобщенного типа объявляется после обобщенного типа-параметра. В этом примере тип T должен поддерживать интерфейс IExpression

Таблица Б.5. Приведение типов и охраняющие выражения типов

Пример	Описание

`let x: unknown = 0;`
 	
`let y: number = <number>x;`
	При указании типа внутри <> перед значением компилятор интерпретирует это значение как относящееся к указанному типу. Переменную x можно присвоить переменной y только после того, как явно повторно интерпретировать ее как number

`type Point = {`
`  x: number;`
`  y: number;`
`}`
 	
`function isPoint(p: unknown):`
`  p is Point {`
` return`
`  ((<Point>p).x`
`  !== undefined) &&`
`  ((<Point>p).y`
`  !== undefined);`
`}`
 	
`let p: unknown = { x: 10, y: 10 };`
 	
`if (isPoint(p)) {`
`  //p здесь относится к типу Point`
`  p.x -= 10;`
`}`
	Предикат типа — это булево значение, указывающее, относится ли переменная к определенному типу. Если заново интерпретировать переменную p как относящуюся к типу Point, причем у нее есть члены типа x и y (и ни один из них не undefined), то предикат p is Point будет возвращать true. Внутри оператора if, в котором предикат типа равен true, проверяемое значение автоматически интерпретируется заново как относящееся к этому типу

Влад Ришкуция

Программируй & типизируй

Перевел с английского *И. Пальти*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*С. Давид*
Ведущий редактор	*Н. Гринчик*
Научный редактор	*Ю. Имбро*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*В. Мостипан*
Корректоры	*Е. Павлович, Т. Радецкая*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 03.2021. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 17.03.21. Формат 70×100/16. Бумага офсетная. Усл. п. л. 28,380. Тираж 500. Заказ 0000.

Отпечатано в полном соответствии с качеством предоставленных материалов в ООО «Фотоэксперт».
109316, г. Москва, Волгоградский проспект, д. 42, корп. 5, эт. 1, пом. I, ком. 6.3-23Н.