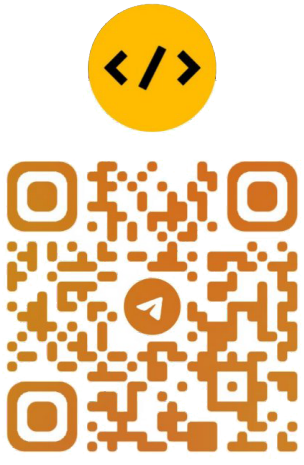
Язык программирования Go

Алан А. А. Донован
Брайан У. Керниган

Серия “Программирование для профессионалов” от издательства Addison-Wesley



@CODELIBRARY_IT

Язык программирования Go

The Go Programming Language

Alan A. A. Donovan
Google Inc.

Brian W. Kernighan
Princeton University

ADDISON-WESLEY

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Язык программирования Go

Алан А. А. Донован
Google Inc.

Брайан У. Керниган
Принстонский университет

Москва • Санкт-Петербург • Киев
2016

ББК 32.973.26-018.2.75
Д67
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция канд. техн. наук. *И.В. Красикова*
Рецензент *Б.У. Керниган*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Донован, Алан А. А., **Керниган**, Брайан, У.

Д67 Язык программирования Go. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2016. — 432 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-2051-5 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Addison-Wesley Publishing Company, Inc.

Authorized translation from the English language edition published by Addison-Wesley Publishing Company, Copyright © 2016 by Alan A. A. Donovan & Brian W. Kernighan.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Back cover: the original Go gopher. © 2009 Renée French. Used under Creative Commons Attributions 3.0 license.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International. Copyright © 2016

Книга отпечатана согласно договору с ООО "Дальрегионсервис".

Научно-популярное издание

Алан А. А. Донован, Брайан У. Керниган

Язык программирования Go

Литературный редактор *Л.Н. Красножон*
Верстка *М.А. Удалов*
Художественный редактор *В.Г. Павлютин*
Корректор *Л.А. Гордиенко*

Подписано в печать 21.03.2016. Формат 70x100/16.
Гарнитура Times. Усл. печ. л. 34,83. Уч.-изд. л. 21,37.
Тираж 700 экз. Заказ № 1872.

Отпечатано способом ролевой струйной печати
в АО «Первая Образцовая типография»
Филиал «Чеховский Печатный Двор»
142300, Московская область, г. Чехов, ул. Полиграфистов, д.1
ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-2051-5 (рус.)
ISBN 978-0-13-419044-0 (англ.)

© Издательский дом "Вильямс", 2016
© Alan A. A. Donovan & Brian W. Kernighan, 2016

Оглавление

Содержание

Предисловие

> "Язык Go является языком программирования с открытым кодом, что делает его простым в создании, надежным и эффективным программным продуктом". (С сайта Go: `golang.org`.)

Go был задуман в сентябре 2007 года Робертом Грисемером (Robert Griesemer), Робом Пайком (Rob Pike) и Кеном Томпсоном (Ken Thompson) из Google и анонсирован в ноябре 2009 года. Целью разработки было создание выразительного, высокоэффективного как при компиляции, так и при выполнении программ языка программирования, позволяющего легко и просто писать надежные высокоинтеллектуальные программы.

Go имеет поверхностное сходство с языком программирования C и обладает тем же духом инструментария для серьезных профессиональных программистов, предназначенного для достижения максимального эффекта с минимальными затратами. Но на самом деле Go — это нечто гораздо большее, чем просто современная версия языка программирования C. Он заимствует и приспосабливает для своих нужд хорошие идеи из многих других языков, избегая возможностей, которые могут привести к созданию сложного и ненадежного кода. Его способности к параллелизму новы и чрезвычайно эффективны, а подход к абстракции данных и объектно-ориентированному программированию непривычный, но необычайно гибкий. Как и все современные языки, Go обладает эффективным механизмом сбора мусора.

Go особенно хорошо подходит для инфраструктуры: построения инструментария и систем для работы других программистов. Однако, будучи в действительности языком общего назначения, он подходит для любого применения и становится все более популярным в качестве замены нетипизированных языков сценариев, обеспечивая компромисс между выразительностью и безопасностью. Программы Go обычно выполняются быстрее, чем программы, написанные на современных динамических языках, и не завершаются аварийно с неожиданными типами ошибок.

Go — это проект с открытым исходным кодом, так что исходные тексты его библиотек и инструментов, включая компилятор, находятся в открытом доступе. Свой вклад в язык, его библиотеки и инструментарий вносят многие программисты всего мира. Go работает на большом количестве Unix-подобных систем, таких как Linux, FreeBSD, OpenBSD, Mac OS X, а также на Plan 9 и Microsoft Windows; при этом программы, написанные для одной из этих сред, легко переносимы на другие.

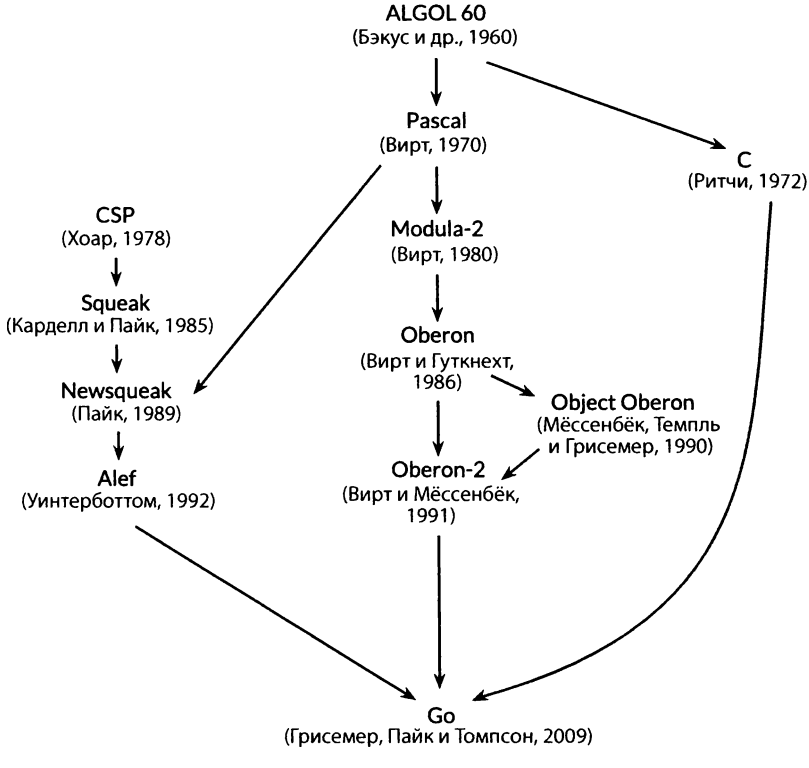
Эта книга призвана помочь вам начать работать с Go, причем с самого начала эффективно использовать все его особенности и богатые стандартные библиотеки для написания понятных, идиоматичных и эффективных программ.

Происхождение Go

Подобно биологическим видам, успешные языки порождают потомство, которое наследует наилучшие особенности своих предков. Скрещивание при этом иногда приводит к удивительным результатам. Аналогом мутаций служит появление радикально новых идей. Как и в случае с живыми существами, глядя на такое влияние предков, можно многое сказать о том, почему язык получился именно таким, какой он есть, и для каких условий работы он приспособлен более всего.

Если вы просто хотите быстро выучить язык, этот раздел является для вас необязательным, но для глубокого понимания Go имеет смысл ознакомиться с его происхождением.

На рисунке ниже показано, какие языки повлияли на дизайн языка программирования Go.



Go часто описывают как "C-подобный язык" или "язык C XXI века". От языка C Go унаследовал синтаксис выражений, конструкции управления потоком, базовые типы данных, передачу параметров в функции по значению, понятие указателей и, что важнее всего, направленность C на получение при компиляции эффективного машинного кода и естественное взаимодействие с абстракциями современных операционных систем.

Однако в генеалогическом древе Go есть и другие предки. Одно из сильнейших влияний на Go оказали языки программирования Никлауса Вирта (Niklaus Wirth), начиная с Pascal. Modula-2 привнесла концепцию пакетов; Oberon использует один файл для определения модуля и его реализации; Oberon-2 явился источником синтаксиса пакетов, импорта и объявлений (прежде всего, объявлений методов), которые он, в свою очередь, унаследовал от языка Object Oberon.

Еще одна линия предков Go, которая выделяет его среди прочих современных языков программирования, представляет собой последовательность малоизвестных исследовательских языков, разработанных в Bell Labs и основанных на концепции *взаимодействующих последовательных процессов* (communicating sequential processes — CSP) из статьи Тони Хоара (Tony Hoare) 1978 года, посвященной основам параллелизма.

В CSP программа представляет собой параллельное объединение процессов, не имеющих общего состояния; процессы взаимодействуют и синхронизируются с помощью каналов. Но CSP Хоара представлял собой формальный язык описания фундаментальных концепций параллелизма, а не язык программирования для написания выполнимых программ.

Роб Пайк (Rob Pike) и другие начали экспериментировать с реализациями CSP в виде фактических языков программирования. Первый из них назывался "Squeak" ("язык для общения с мышью"), который являлся языком для обработки событий мыши и клавиатуры со статически созданными каналами. За ним последовал Newsqueak, в котором C-образные инструкции и синтаксис выражений сочетались с записью типов в стиле Pascal. Это был чисто функциональный язык со сборкой мусора, направленный, как и его предшественник, на управление клавиатурой, мышью и оконными событиями. Каналы в нем стали полноправными участниками языка, динамически создаваемыми и хранимыми в переменных.

Операционная система Plan 9 развила эти идеи в языке Alef. Alef попытался сделать Newsqueak жизнеспособным системным языком программирования, но параллелизм без сборки мусора требовал слишком больших усилий.

Ряд конструкций в Go демонстрирует влияние генов непрямых предков; например, `iota` происходит из APL, а лексическая область видимости с вложенными функциями — из Scheme (и большинства последующих за ним языков). Здесь мы находим и признаки мутации: инновационные элементы Go предоставляют динамические массивы с эффективным произвольным доступом, но при этом разрешают сложные размещения, напоминающие связанные списки. Инструкция `defer` также представляет собой новинку Go.

Проект Go

Все языки программирования тем или иным образом отражают философию их создателей, что часто приводит к включению в язык программирования их реакции на слабости и недостатки более ранних языков. Go не является исключением. Проект

Go родился из разочарования в Google несколькими программными системами, страдающими от "взрыва сложности" (эта проблема отнюдь не уникальна для Google).

Как заметил Роб Пайк (Rob Pike), "сложность мультипликативна": устранение проблемы путем усложнения одной части системы медленно, но верно добавляет сложность в другие части. Постоянное требование внесения новых функций, настроек и конфигураций очень быстро заставляет отказаться от простоты, несмотря на то что в долгосрочной перспективе простота является ключом к хорошему программному обеспечению.

Простота требует большего количества работы в начале проекта по определению самой сути идеи и большей дисциплины во время жизненного цикла проекта, которая поможет отличать хорошие изменения от плохих. При достаточных усилиях хорошие изменения, в отличие от плохих, могут быть приняты без ущерба для того, что Фред Брукс (Fred Brooks) назвал "концептуальной целостностью" проекта. Плохие же изменения всего лишь разменивают простоту на удобство. Только с помощью простоты дизайна система может в процессе роста оставаться устойчивой, безопасной и последовательной. Проект Go включает в себя сам язык, его инструментарий, стандартные библиотеки и последнее (по списку, но не по значению) — культуру радикальной простоты. Будучи одним из современных высокоуровневых языков, Go обладает преимуществом ретроспективного анализа других языков, и это преимущество использовано в полной мере: в Go имеются сборка мусора, система пакетов, полноценные функции, лексическая область видимости, интерфейс системных вызовов и неизменяемые строки, текст в которых кодируется с использованием кодировки UTF-8. Однако язык программирования Go имеет сравнительно немного возможностей, и вряд ли в него будут добавлены новые. Например, в нем отсутствуют неявные числовые преобразования, нет конструкторов и деструкторов, перегрузки операторов, значений параметров по умолчанию; нет наследования, обобщенных типов, исключений; отсутствуют макросы, аннотации функций и локальная память потока. Язык программирования Go является зрелым и стабильным и гарантирует обратную совместимость: старые программы на Go можно компилировать и запускать с помощью новых версий компиляторов и стандартных библиотек.

Go имеет достаточную систему типов, чтобы избежать большинства ошибок программистов в динамических языках, но эта система типов гораздо более ограниченная, чем в других строго типизированных языках программирования. Это приводит к изолированным очагам "нетипизированного программирования"' в пределах более широких схем типов. Так что программисты на Go не прибегают к длинным конструкциям, которыми программисты на C++ или Haskell пытаются выразить свойства безопасности своих программ на основе типов. На практике Go дает программистам преимущества безопасности и производительности времени выполнения относительно строгой системы типов без излишней сложности и накладных расходов.

Go поощряет понимание дизайна современных компьютерных систем, в частности — важность локализации. Его встроенные типы данных и большинство библиотечных структур данных созданы для естественной работы без явной инициализации или неявных конструкторов, так что в коде скрывается относительно мало распределений и записей памяти. Составные типы Go (структуры и массивы) хранят свои

элементы непосредственно, требуя меньшего количества памяти и ее распределений, а также меньшего количества косвенных обращений с помощью указателей по сравнению с языками, использующими косвенные поля. А поскольку современные компьютеры являются параллельными вычислительными машинами, Go обладает возможностями параллельности, основанными, как упоминалось ранее, на CSP. Стеки переменного размера легких потоков (или *go-подпрограмм* (goroutines)) Go изначально достаточно малы, чтобы создание одной go-подпрограммы было дешевым, а создание миллиона — практичным.

Стандартная библиотека Go, часто описываемая фразой "все включено", предоставляет строительные блоки и API для ввода-вывода, работы с текстом и графикой, криптографические функции, функции для работы с сетью и для создания распределенных приложений. Библиотека поддерживает множество стандартных форматов файлов и протоколов. Библиотеки и инструменты интенсивно используют соглашения по снижению потребностей в настройке, упрощая тем самым логику программ; таким образом, различные программы Go становятся более похожими одна на другую и тем самым — более простыми в изучении. Проекты создаются с помощью всего лишь одного инструмента `go` и используют только имена файлов и идентификаторов и иногда — специальные комментарии для определения всех библиотек, выполнимых файлов, тестов, примеров, документации и прочего в проектах; исходный текст Go содержит всю необходимую спецификацию построения проекта.

Структура книги

Мы предполагаем, что читатель программирует на одном или нескольких современных языках программирования, компилирующих языках наподобие C, C++ и Java или динамических, таких как Python, Ruby и JavaScript. Таким образом, мы не стараемся излагать материал так, как будто имеем дело с новичками в программировании. Внешне синтаксис будет вам знаком, так как будет содержать переменные и константы, выражения, управление потоком и функции.

Глава 1 представляет собой руководство по базовым конструкциям Go, содержащее массу небольших программ для решения ежедневных задач наподобие чтения и записи файлов, форматированного вывода результатов, соединений "клиент/сервер" в Интернете и т.п.

В главе 2 описаны структурные элементы программы Go — объявления, переменные, новые типы, пакеты и файлы, области видимости. В главе 3 рассмотрены основные типы данных — числа, логические значения, строки и константы. В главе 4 изучаются составные типы, т.е. типы, построенные из более простых типов с помощью таких механизмов, как массивы, отображения, структуры, а также *срезы* (slices) — нетрадиционное представление динамических списков в Go. Глава 5 посвящена функциям, обработке ошибок, а также инструкциям `panic`, `recover` и `defer`.

Таким образом, главы 1–5 представляют собой реальную основу, то, что является частью любого императивного языка. Синтаксис и стиль Go иногда отличаются от привычных для других языков программирования, но большинство программистов

быстро к этому привыкают. В остальных главах внимание сосредоточено на темах, в которых подход Go менее привычен: методы, интерфейсы, параллелизм, пакеты, тестирование и рефлексия.

Go демонстрирует нестандартный подход к объектно-ориентированному программированию: в нем нет ни иерархий классов, ни каких-либо классов; методы могут быть связаны с любым типом, а не только со структурами; поведение сложных объектов создается из более простых не путем наследования, а с помощью композиции; наконец взаимосвязь между конкретными типами и абстрактными типами (*интерфейсами*) является неявной, так что конкретный тип может удовлетворять интерфейсу так, что его разработчик не будет об этом знать. Методы описаны в главе 6, а интерфейсы — в главе 7.

Go обеспечивает эффективный и удобный механизм параллелизма, используя *go-подпрограммы* (goroutines) и каналы и основываясь на упомянутых выше идеях взаимодействующих последовательных процессов. Широко известные возможности параллелизма Go являются темой главы 8. В главе 9 рассмотрены более традиционные аспекты параллелизма на основе совместно используемых переменных.

В главе 10 описаны пакеты, представляющие собой механизм для организации библиотек. В этой главе также показано, как эффективно использовать инструмент `go`, который обеспечивает компиляцию, тестирование, форматирование программы, документирование и другие задачи, все — единой командой.

Глава 11 посвящена тестированию, где Go использует особенно простой подход, избегая абстрактных схем и предпочитая простые библиотеки и инструменты. Библиотеки тестирования предоставляют основу, на которой при необходимости могут быть построены более сложные абстракции.

В главе 12 рассматривается рефлексия — возможность получения программой информации о собственном представлении во время выполнения. Рефлексия является мощным инструментом, но использовать его следует с осторожностью. В этой главе речь идет о том, как найти правильный компромисс, на примере применения рефлексии при реализации некоторых важных библиотек Go. Глава 13 касается деталей низкоуровневого программирования: как использовать пакет `unsafe` для обхода системы типов Go (и когда это уместно).

В каждой главе имеется ряд упражнений, которые можно использовать для проверки своего понимания Go и для изучения расширений и альтернатив примерам из книги.

Все примеры кода, кроме самых тривиальных, в книге доступны для загрузки из репозитория `git` по адресу `gopl.io`. Каждый пример идентифицируется своим путем импорта пакетов и может быть легко получен с помощью команды `go get`. Вам нужно выбрать каталог, который будет использоваться в качестве рабочего пространства Go, и установить переменную среды `GOPATH` так, чтобы она указывала на этот каталог. Инструмент `go` при необходимости сам создаст этот каталог. Например:

```
$ export GOPATH=$HOME/gobook     # Каталог рабочего пространства
$ go get gopl.io/ch1/helloworld # Выборка, построение, установка
$ $GOPATH/bin/helloworld         # Запуск
Hello, World
```

Для выполнения примеров нужна версия Go не ниже 1.5.

```
$ go version
go version go1.5 linux/amd64
```

Если ваша версия не отвечает указанному требованию, следуйте инструкциям, расположенным по адресу `https://golang.org/doc/install`.

Дополнительная информация

Основным источником дополнительной информации о Go является официальный веб-сайт `https://golang.org`. Он предоставляет доступ к документации, включая официальную спецификацию языка Go, к стандартным пакетам и т.п. Здесь, кроме того, есть учебные пособия о том, как писать на Go и писать хорошо, а также широкий спектр текстовых и видеоресурсов. Все это будет ценным дополнением к данной книге. В блоге Go по адресу `blog.golang.org` публикуются некоторые из лучших материалов о Go, включая статьи о состоянии языка, планы на будущее, доклады на конференциях и углубленное объяснение широкого спектра тем, связанных с Go.

Одна из самых полезных возможностей онлайн-доступа к Go (и, к сожалению, напрочь отсутствующая в бумажной книге) — возможность запускать программы Go с веб-страниц, на которых эти программы описаны. Эта функциональность обеспечивается сайтом Go Playground по адресу `play.golang.org` и может быть встроена в другие страницы, как, например, начальная страница `golang.org` или страница документации, обслуживаемая инструментом `godoc`.

Go Playground позволяет проводить простые эксперименты с короткими программами для проверки понимания синтаксиса, семантики или библиотечных пакетов. Постоянный URL позволяет обмениваться фрагментами кода Go с другими программистами, сообщать об ошибках или вносить свои предложения. Авторы используют этот сайт по многу раз в день.

Будучи надстройкой над Go Playground, Go Tour по адресу `tour.golang.org` представляет собой последовательность около 75 коротких интерактивных уроков, посвященных основным идеям и конструкциям языка программирования Go и упорядоченных в виде обучающего курса по языку Go.

Основным недостатком Go Playground и Go Tour является то, что они позволяют импортировать только стандартные библиотеки, и к тому же многие библиотечные функции — например, сетевые — ограничены по практическим соображениям и для большей безопасности. Кроме того, для компиляции и выполнения каждой программы они требуют доступа к Интернету. Таким образом, для более сложных экспериментов вам придется запускать программы Go на своем компьютере. К счастью, процесс загрузки прост, и, как отмечалось выше, Go работает в любой современной операционной системе. Так что загрузка Go с сайта `golang.org` и установка на компьютере не должна занять более нескольких минут, после чего вы сможете начать написание и запуск программ Go самостоятельно, не будучи привязанными к конкретному сайту.

Поскольку Go представляет собой проект с открытым кодом, вы можете прочесть код любого типа или функции в стандартной библиотеке, доступной онлайн по адресу `https://golang.org/pkg`; этот же код является частью загружаемого дистрибутива. Используйте это, чтобы понять, как работает та или иная возможность или функция, чтобы получить подробные ответы на свои вопросы или просто узнать, как выглядит хороший код на Go, написанный экспертами.

Благодарности

Роб Пайк (Rob Pike) и Расс Кокс (Russ Cox), одни из лидеров команды Go, неоднократно тщательно читали рукопись; их комментарии касались всего — от выбора слов до общей структуры и организации книги — и были поистине бесценны. При подготовке японского перевода Ёсики Сибата (Yoshiki Shibata) вышел далеко за рамки служебного долга; его дотошный взгляд обнаружил ошибки в коде и многочисленные несоответствия в английском тексте. Мы высоко ценим тщательные обзоры и критические замечания по данной рукописи, сделанные Брайаном Гётцем (Brian Goetz), Кори Косак (Corey Kosak), Арнольдом Роббинсом (Arnold Robbins), Джошем Бличером Снайдером (Josh Bleecher Snyder) и Питером Вайнбергером (Peter Weinberger).

Мы в долгу перед Самиром Аджмани (Sameer Ajmani), Иттаи Балабаном (Ittai Balaban), Дэвидом Кроушоу (David Crawshaw), Билли Донохью (Billy Donohue), Джонатаном Файнбергом (Jonathan Feinberg), Эндрю Жеррандом (Andrew Gerrand), Робертом Грисемером (Robert Griesemer), Джоном Линдерманом (John Linderman), Минуксом Ма (Minux Ma), Брайаном Миллсом (Bryan Mills), Балой Натаражан (Bala Natarajan), Космосом Николау (Cosmos Nicolaou), Полом Станифортом (Paul Staniforth), Нигелем Тао (Nigel Tao) и Говардом Трики (Howard Trickey) за множество полезных предложений. Мы также благодарим Дэвида Брейлсфорда (David Brailsford) и Рафа Левина (Raph Levien) за советы при верстке.

Наш редактор Грег Донч (Greg Doench) из Addison-Wesley был неизменно полезен — с момента получения этой книги в работу. Издательская команда — Джон Фуллер (John Fuller), Дайна Исли (Dayna Isley), Джули Нагил (Julie Nahil), Чути Прасертсих (Chuti Prasertsith) и Барбара Вуд (Barbara Wood) — оказалась выдающейся; мы не могли и надеяться на лучшее.

Алан Донован (Alan Donovan) благодарит Самира Аджмани (Sameer Ajmani), Криса Деметроу (Chris Demetriou), Уолта Драммонда (Walt Drummond) и Рейда Татджа (Reid Tatge) из Google за то, что они помогли ему найти время для написания книги; Стивена Донована (Stephen Donovan) — за его советы и своевременную поддержку; а больше всех — его жену Лейлу Каземи (Leila Kazemi) за ее энтузиазм и неизменную поддержку этого проекта, несмотря на то что он надолго отвлекал ее мужа от семьи.

Брайан Керниган (Brian Kernighan) глубоко признателен друзьям и коллегам за их терпение и выдержку, а в особенности — своей жене Мэг, которая поддерживала его как при написании книги, так и во всем другом.

Нью-Йорк
Октябрь 2015

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: `info@williamspublishing.com`

WWW: `http://www.williamspublishing.com`

Наши почтовые адреса:

в России: 127055, Москва, ул. Лесная, д. 43, стр. 1

в Украине: 03150, Киев, а/я 152

1

Учебник

Эта глава представляет собой краткий обзор основных компонентов Go. Здесь предоставлено достаточно информации и примеров для того, чтобы вы научились делать полезные вещи с помощью Go как можно быстрее. Приведенные здесь (как и во всей книге) примеры позволяют решать задачи, которые вам, возможно, придется или приходилось решать в реальной практике программиста. В этой главе мы имеем дело с простыми программами, которые читают ввод, что-то с ним делают и записывают выходные данные. Такие программы должны быть знакомы вам из вашей практики на других языках программирования, так что они являются очень эффективным средством для начала изучения нового языка.

При изучении нового языка программирования существует естественная тенденция записывать новый код так, как вы записали бы его на хорошо знакомом вам языке (или, как метко было сказано еще до рождения большинства читателей этой книги, "писать программу на Fortran с точками с запятой"). Попытайтесь противостоять этой тенденции. В книге мы постарались показать и объяснить, как писать хорошие идиоматичные программы на Go, так что используйте представленный здесь код как руководство при написании собственного кода.

1.1. Hello, World

Давайте начнем с традиционной программы "hello, world", которая впервые появилась в книге *The C Programming Language* в 1978 году. Поскольку язык C — один из основных прямых предков Go, у нас есть исторические основания для того, чтобы начать с данной программы; кроме того, она иллюстрирует ряд центральных идей.

gopl.io/ch1/helloworld

```
package main
import "fmt"

func main() {
    fmt.Println("Hello, мир")
}
```

Go — компилируемый язык. Его инструментарий преобразует исходный текст программы, а также библиотеки, от которых он зависит, в команды на машинном

языке компьютера. Доступ к инструментарию Go осуществляется с помощью единой команды `go`, которая имеет множество подкоманд. Простейшей из них является подкоманда `run`, компилирующая исходный текст из одного или нескольких исходных файлов с расширением `.go`, связывает их с библиотеками, а затем выполняет полученный в результате выполнимый файл. (Здесь и далее в книге символ `$` используется в качестве приглашения командной строки.)

```
$ go run helloworld.go
```

Ничего удивительного, что программа выводит строку

```
Hello, мир
```

Go изначально обрабатывает символы Unicode, так что он может обработать текст с любым набором символов.

Если программа предназначена более чем для разового эксперимента, вероятно, вы захотите скомпилировать ее и сохранить скомпилированный результат для дальнейшего использования. Это делается с помощью команды `go build`:

```
$ go build helloworld.go
```

Она создает бинарный выполнимый файл `helloworld`, который можно выполнить в любой момент времени без какой-либо обработки:

```
$ ./helloworld
Hello, мир
```

Каждый важный пример мы помечаем напоминанием о том, что вы можете получить соответствующий исходный текст из репозитория исходных текстов данной книги по адресу `gopl.io`:

gopl.io/ch1/helloworld

Если вы выполните команду `go get gopl.io/ch1/helloworld`, она сделает выборку необходимого исходного текста и поместит его в соответствующий каталог. Более подробно об этом речь идет в разделах 2.6 и 10.7.

Давайте теперь поговорим о самой программе. Код Go организован в виде пакетов, которые подобны библиотекам или модулям других языков. Пакет состоит из одного или нескольких файлов исходных текстов (`.go`) в одном каталоге, которые определяют, какие действия выполняет данный пакет. Каждый исходный файл начинается с объявления `package` (в данном случае — `package main`), которое указывает, к какому пакету принадлежит данный файл. Затем следует список других пакетов, которые этот файл импортирует, а после него — объявления программы, хранящейся в этом исходном файле.

Стандартная библиотека Go имеет более сотни пакетов для распространенных задач, таких как ввод и вывод, сортировка, работа с текстом и т.д. Например, пакет `fmt` содержит функции для форматированного вывода и сканирования ввода. Функция `Println` является одной из основных функций в пакете `fmt`; она выводит одно или несколько значений, разделенных пробелами и с добавлением символа новой строки в конце, так что выводимые значения располагаются в одной строке.

Пакет `main` — особый. Он определяет отдельную программу, т.е. выполнимый файл, а не библиотеку. В пакете `main` *функция* `main` также является особой — именно с нее начинается программа. Программа делает то, что делается в функции `main`. Конечно, для выполнения действий функция `main` обычно вызывает функции из других пакетов, как, например, `fmt.Println`.

Мы должны указать компилятору, какие пакеты необходимы для данного исходного файла; эту задачу решают объявления `import`, следующие за объявлением `package`. Программа "hello, world" использует только одну функцию из одного стороннего пакета, но большинство программ импортируют несколько пакетов.

Необходимо импортировать только те пакеты, которые вам нужны. Программа не будет компилироваться как при наличии отсутствующего импорта пакетов, так и при наличии излишнего. Это строгое требование предотвращает накопление ссылок на неиспользуемые пакеты по мере развития программы.

Объявления `import` должны следовать за объявлением `package`.

После этого в программе располагаются объявления функций, переменных, констант и типов (вводимые с помощью ключевых слов `func`, `var`, `const` и `type`); по большей части порядок объявлений не имеет значения. Приведенная программа максимально короткая, так как объявляет только одну функцию, которая, в свою очередь, также вызывает только одну функцию. Для экономии места мы не всегда будем указывать объявления `package` и `import` в примерах, но они имеются в исходных файлах (и должны быть там, чтобы код успешно компилировался).

Объявление функции состоит из ключевого слова `func`, имени функции, списка параметров (пустого списка в случае функции `main`), списка результатов (также пустого в данном случае) и тела функции — инструкций, которые определяют выполняемые функцией действия — в фигурных скобках. Более подробно функции рассматриваются в главе 5, "Функции".

Go не требует точек с запятой в конце инструкции или объявления, за исключением случаев, когда две или более инструкций находятся в одной и той же строке. По сути, символы новой строки после определенных лексем преобразуются в точки с запятой, так что местоположение символов новой строки имеет значение для корректного синтаксического анализа кода Go. Например, открывающая фигурная скобка { функции должна находиться в той же строке, что и конец объявления `func`, но не в отдельной строке, а в выражении `x + y` символ новой строки разрешен после, но не до оператора +.

Go занимает жесткую позицию относительно форматирования кода. Инструмент `gofmt` приводит код к стандартному формату, а подкоманда `fmt` инструмента `go` применяет `gofmt` ко всем файлам в указанном пакете или к файлам в текущем каталоге по умолчанию. Все исходные файлы Go в книге пропущены через `gofmt`, и вам нужно выработать привычку поступать так же со своим кодом. Формальное объявление стандартного формата устраняет множество бессмысленных споров о мелочах и, что более важно, разрешает целый ряд автоматизированных преобразований исходного кода, которые были бы неосуществимы при разрешенном произвольном форматировании.

Многие текстовые редакторы могут настраиваться так, что при каждом сохранении файла будет запускаться инструмент `gofmt`, так что ваш исходный текст всегда будет правильно отформатирован. Еще один инструмент, `goimports`, дополнительно управляет вставкой и удалением объявлений импорта при необходимости. Он не является частью стандартного дистрибутива, но вы можете получить его с помощью следующей команды:

```
$ go get golang.org/x/tools/cmd/goimports
```

Для большинства пользователей обычным средством загрузки и построения пакетов, запуска тестов, показа документации и так далее является инструмент `go`, который мы рассмотрим в разделе 10.7.

1.2. Аргументы командной строки

Большинство программ обрабатывают некоторые входные данные для генерации некоторых выходных данных; это довольно точное определение вычислений. Но как получить входные данные для работы программы? Некоторые программы генерируют собственные данные, но чаще ввод поступает из внешнего источника: файла, подключения к сети, вывода из другой программы, ввода пользователя с помощью клавиатуры, из аргументов командной строки или другого подобного источника. В нескольких следующих примерах мы рассмотрим некоторые из перечисленных альтернатив, начиная с аргументов командной строки.

Пакет `os` предоставляет функции и различные значения для работы с операционной системой платформо-независимым образом. Аргументы командной строки доступны в программе в виде переменной с именем `Args`, которая является частью пакета `os` (таким образом, ее имя в любом месте за пределами пакета `os` выглядит как `os.Args`).

Переменная `os.Args` представляет собой *срез* (slice) строк. Срезы являются фундаментальным понятием в Go, и вскоре мы поговорим о них гораздо подробнее. Пока же думайте о срезе как о последовательности (с динамическим размером) `s` элементов массива, в которой к отдельным элементам можно обратиться как к `s[i]`, а к непрерывной подпоследовательности — как к `s[m:n]`. Количество этих элементов определяется как `len(s)`. Как и в большинстве других языков программирования, индексация в Go использует *полуоткрытые* интервалы, которые включают первый индекс, но исключают последний, потому что это упрощает логику. Например, срез `s[m:n]`, где $0 \le m \le n \le$ `len(s)`, содержит `n-m` элементов.

Первый элемент `os.Args`, `os.Args[0]`, представляет собой имя самой команды; остальные элементы представляют собой аргументы, которые были переданы программе, когда началось ее выполнение. Выражение вида `s[m:n]` дает срез, который указывает на элементы от `m` до `n-1`-го, так что элементы, которые нам потребуются в следующем примере, находятся в срезе `os.Args[1:len(os.Args)]`. Если опущено значение `m` или `n`, используются значения по умолчанию — `0` или `len(s)` соответственно, так что мы можем сократить запись нужного нам среза до `os.Args[1:]`.

Далее представлена реализация команды Unix `echo`, которая выводит в одну строку аргументы, переданные в командной строке. Она импортирует два пакета, которые указаны в круглых скобках, а не в виде отдельных объявлений импорта. Можно использовать любую запись, но обычно используется список. Порядок импорта значения не имеет; инструмент `gofmt` сортирует имена пакетов в алфавитном порядке. (Если имеется несколько версий примера, мы будем часто их нумеровать, чтобы вы точно знали, о какой из них мы говорим.)

gopl.io/ch1/echo1

```
// Echo1 выводит аргументы командной строки
package main

import (
    "fmt"
    "os"
)
func main() {
    var s, sep string
    for i := 1; i < len(os.Args); i++ {
        s += sep + os.Args[i]
        sep = " "
    }
    fmt.Println(s)
}
```

Комментарии начинаются с символов `//`. Весь текст от `//` до конца строки является комментарием, предназначенным для программиста, и игнорируется компилятором. По соглашению мы описываем каждый пакет в комментарии, непосредственно предшествующем его объявлению; для пакета `main` этот комментарий состоит из одного или нескольких полных предложений, которые описывают программу в целом.

Объявление `var` объявляет две переменные — `s` и `sep` типа `string`. Как часть объявления переменная может быть инициализирована. Если переменная не инициализирована явно, она неявно инициализируется *нулевым значением* соответствующего типа (которое равно `0` для числовых типов и пустой строке `""` для строк). Таким образом, в этом примере объявление неявно инициализирует строки `s` и `sep`. Более подробно о переменных и объявлениях мы поговорим в главе 2, "Структура программы".

Для чисел Go предоставляет обычные арифметические и логические операторы. Однако при применении к строкам оператор + выполняет *конкатенацию* их значений, так что выражение

```
sep + os.Args[i]
```

представляет собой конкатенацию строк `sep` и `os.Args[i]`. Использованная в программе инструкция

```
s += sep + os.Args[i]
```

представляет собой *инструкцию присваивания*, которая выполняет конкатенацию старого значения `s` с `sep` и `os.Args[i]` и присваивает новое значение переменной `s`; она эквивалентна выражению

```
s = s + sep + os.Args[i]
```

Оператор `+=` является *присваивающим оператором*. Каждый арифметический и логический оператор наподобие `+` или `*` имеет соответствующий присваивающий оператор.

Программа `echo` могла бы вывести все выходные данные в цикле по одному фрагменту за раз, но наша версия вместо этого строит строку, добавляя новый текст в конце каждого фрагмента. Изначально строка `s` пуста, т.е. имеет значение `""`, и каждая итерация цикла добавляет к ней текст. После первой итерации перед очередным фрагментом вставляется пробел, так что после завершения цикла между всеми аргументами имеются пробелы. Этот процесс имеет квадратичное время работы, так что он может оказаться дорогостоящим, если количество аргументов будет большим, но для `echo` это маловероятно. Далее в этой и следующей главах мы приведем несколько усовершенствованных версий `echo`.

Индексная переменная цикла `i` объявлена в первой части цикла `for`. Символы `:=` являются частью *краткого объявления переменной*, инструкции, которая объявляет одну или несколько переменных и назначает им соответствующие типы на основе значения инициализатора. В следующей главе мы расскажем об этом подробнее.

Оператор инкремента `i++` добавляет к `i` единицу. Эта запись эквивалентна записи `i += 1`, которая, в свою очередь, эквивалентна записи `i = i + 1`. Имеется и соответствующий оператор декремента, который вычитает `1`. Это инструкции, а не выражения, как в большинстве языков семейства C, поэтому запись `j = i++` является некорректной; кроме того, эти операторы могут быть только постфиксными.

Цикл `for` является единственной инструкцией цикла в Go. Он имеет ряд разновидностей, одна из которых показана здесь:

```
for инициализация; условие; последействие {
    // нуль или несколько инструкций
}
```

Вокруг трех компонентов цикла `for` скобки не используются. Фигурные же скобки обязательны, причем открывающая фигурная скобка обязана находиться в той же строке, что и инструкция *последействие*.

Необязательная инструкция *инициализации* выполняется до начала работы цикла. Если она имеется в наличии, она обязана быть *простой инструкцией*, т.е. кратким объявлением переменной, инструкцией инкремента или присваивания, или вызовом функции.

Условие представляет собой логическое выражение, которое вычисляется в начале каждой итерации цикла. Если его вычисление дает результат `true`, выполняются инструкции тела цикла. Инструкция *последействие* выполняется после тела цикла, после чего вновь вычисляется *условие*. Цикл завершается, когда вычисление условия дает значение `false`.

Любая из перечисленных частей может быть опущена. При отсутствии как *инициализации*, так и *последействия* можно опустить точки с запятыми:

```
// Традиционный цикл "while"
for condition {
    // ...
}
```

Если условие опущено полностью в любой из разновидностей цикла, например, в

```
// Традиционный бесконечный цикл
for {
    // ...
}
```

мы получаем бесконечный цикл, который должен завершиться некоторым иным путем, например с помощью инструкции `break` или `return`.

Еще одна разновидность цикла `for` выполняет итерации для *диапазона* значений для типа данных наподобие строки или среза. Для иллюстрации приведем вторую версию программы `echo`:

gopl.io/ch1/echo2

```
// Echo2 выводит аргументы командной строки.
package main

import (
    "fmt"
    "os"
)

func main() {
    s, sep := "", ""
    for _, arg := range os.Args[1:] {
        s += sep + arg
        sep = " "
    }
    fmt.Println(s)
}
```

В каждой итерации цикла `range` производит пару значений: индекс и значение элемента с этим индексом. В данном примере мы не нуждаемся в индексе, но синтаксис цикла по диапазону требует, чтобы, имея дело с элементом, мы работали и с индексом. Одно из решений заключается в том, чтобы присваивать значение индекса временной переменной с очевидным именем наподобие `temp` и игнорировать его. Однако Go не допускает наличия неиспользуемых локальных переменных, так что этот способ приведет к ошибке компиляции.

Решение заключается в применении *пустого идентификатора* (blank identifier) с именем _ (символ подчеркивания). Пустой идентификатор может использоваться везде, где синтаксис требует имя переменной, но логике программы он не нужен, напри-

мер в цикле по диапазону, в котором нам достаточно знать значение элемента, но не его индекс. Большинство программистов Go предпочтет использовать `range` и `_` для записи программы `echo` (как это сделано выше), поскольку индексирование `os.Args` выполняется при этом неявно, а значит, труднее допустить ошибку при написании.

В этой версии программы для объявления и инициализации `s` и `sep` используется краткое объявление переменной, но мы можем объявить эти переменные и отдельно. Имеются разные способы объявления строковых переменных; приведенные далее объявления эквивалентны:

```
s := ""
var s string
var s = ""
var s string = ""
```

Почему мы должны предпочитать один вид объявления другим? Первая разновидность, краткое объявление переменной, является наиболее компактной, однако может использоваться только внутри функции, но не для переменных уровня пакета. Вторая разновидность основана на инициализации по умолчанию (для строки — значением `""`). Третья разновидность используется редко, в основном при объявлении нескольких переменных. Четвертая разновидность содержит явное указание типа переменной, которое является излишним, когда тип совпадает с типом начального значения переменной, но которое является обязательным в других случаях, когда типы переменной и инициализатора разные. На практике обычно следует использовать одну из первых двух разновидностей: с явной инициализацией (чтобы указать важность начального значения) и с неявной инициализацией по умолчанию (чтобы указать, что начальное значение не играет роли).

Как отмечалось выше, при каждой итерации цикла строка `s` получает новое содержимое. Оператор `+=` создает новую строку путем конкатенации старой строки, символа пробела и очередного аргумента, а затем присваивает новую строку переменной `s`. Старое содержимое строки `s` более не используется, поэтому оно будет в надлежащее время обработано сборщиком мусора.

Если объем данных велик, это может быть дорогостоящим решением. Более простым и более эффективным решением было бы использование функции `Join` из пакета `strings`:

gopl.io/ch1/echo3

```
func main() {
    fmt.Println(strings.Join(os.Args[1:], " "))
}
```

Наконец, если нам не нужно беспокоиться о формате и нужно увидеть только значения, например, для отладки, мы можем позволить функции `Println` форматировать результаты для нас:

```
fmt.Println(os.Args[1:])
```

Вывод этой инструкции похож на вывод, полученный в версии с применением `strings.Join`, но с окружающими квадратными скобками. Таким образом может быть выведен любой срез.

Упражнение 1.1. Измените программу `echo` так, чтобы она выводила также `os.Args[0]`, имя выполняемой команды.

Упражнение 1.2. Измените программу `echo` так, чтобы она выводила индекс и значение каждого аргумента по одному аргументу в строке.

Упражнение 1.3. Поэкспериментируйте с измерением разницы времени выполнения потенциально неэффективных версий и версии с применением `strings.Join`. (В разделе 1.6 демонстрируется часть пакета `time`, а в разделе 11.4 — как написать тест производительности для ее систематической оценки.)

1.3. Поиск повторяющихся строк

Программы для копирования файлов, печати, поиска, сортировки, подсчета и другие имеют схожую структуру: цикл по входным данным, некоторые вычисления над каждым элементом и генерация вывода "на лету" или по окончании вычислений. Мы покажем три варианта программы под названием "dup", на создание которой нас натолкнула команда `uniq` из Unix, которая ищет соседние повторяющиеся строки. Использованные структуры и пакеты представляют собой модели, которые могут быть легко адаптированы.

Первая версия `dup` выводит каждую строку, которая в стандартном вводе появляется больше одного раза, выводя предварительно количество ее появлений. В этой программе вводятся инструкция `if`, тип данных `map` и пакет `bufio`.

gopl.io/ch1/dup1

```
// Dup1 выводит текст каждой строки, которая появляется в
// стандартном вводе более одного раза, а также количество
// ее появлений.
package main

import (
    "bufio"
    "fmt"
    "os"
)

func main() {
    counts := make(map[string]int)
    input := bufio.NewScanner(os.Stdin)
    for input.Scan() {
        counts[input.Text()]++
    }
    // Примечание: игнорируем потенциальные
    // ошибки из input.Err()
    for line, n := range counts {
```

```
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}
```

Как и в цикле `for`, вокруг условия инструкции `if` нет скобок, но для тела инструкции фигурные скобки обязательны. Может иметься необязательная часть `else`, которая выполняется при ложности условия.

Отображение (map) содержит набор пар "ключ–значение" и обеспечивает константное время выполнения операций хранения, извлечения или проверки наличия элемента в множестве. Ключ может быть любого типа, лишь бы значения этого типа можно было сравнить с помощью оператора `==`; распространенным примером ключа являются строки. Значение может быть любого типа. В нашем примере ключи представляют собой строки, а значения представлены типом `int`. Встроенная функция `make` создает новое пустое отображение; она имеет и другие применения. Отображения подробно обсуждаются в разделе 4.3.

Всякий раз, когда `dup` считывает строку ввода, эта строка используется как ключ в отображении, и соответствующее значение увеличивается. Инструкция `counts [input.Text()]++` эквивалентна следующим двум инструкциям:

```
line := input.Text()
counts[line] = counts[line] + 1
```

Если в отображении еще нет нужного нам ключа, это не проблема. Когда новая строка встречается впервые, выражение `counts[line]` в правой части присваивает нулевое значение новому элементу, которое для типа `int` равно `0`.

Для вывода результатов мы вновь используем цикл по диапазону, на этот раз — по отображению `counts`. Как и ранее, каждая итерация дает две величины — ключ и значение элемента отображения для этого ключа. Порядок обхода отображения не определен, на практике этот порядок случаен и варьируется от одного выполнения программы к другому. Это сделано преднамеренно, поскольку предотвращает написание программ, опирающихся на конкретное упорядочение, которое не гарантируется[1].

Далее наступает очередь пакета `bufio`, который помогает сделать ввод и вывод эффективным и удобным. Одной из наиболее полезных его возможностей является тип с именем `Scanner`, который считывает входные данные и разбивает их на строки или слова; зачастую это самый простой способ обработки ввода, который поступает построчно.

Программа использует краткое объявление переменной для создания новой переменной `input`, которая ссылается на `bufio.Scanner`:

```
input := bufio.NewScanner(os.Stdin)
```

Сканер считывает стандартный ввод программы. Каждый вызов `input.Scan()` считывает очередную строку и удаляет завершающий символ новой строки; результат

[1] В первую очередь, отсутствие упорядочения связано с тем, что при упорядочении было бы недостижимо константное время операций с отображением. — *Примеч. ред.*

можно получить путем вызова `input.Text()`. Функция `Scan` возвращает значение `true`, если строка считана и доступна, и значение `false`, если входные данные исчерпаны.

Функция `fmt.Printf`, подобно функции `printf` в языке программирования C и других языках, выполняет форматированный вывод на основе списка выражений. Первым ее аргументом является строка формата, которая указывает, как должны быть отформатированы последующие аргументы. Формат каждого аргумента определяется символом преобразования, буквой, следующей за знаком процента. Например, `%d` форматирует целочисленный операнд в десятичной записи, а `%s` выводит значение строкового операнда.

Функция `Printf` имеет больше десятка таких преобразований, которые программисты на Go называют *глаголами* (verbs). Приведенная далее таблица далека от полной спецификации, но иллюстрирует ряд доступных возможностей.

`%d`	Десятичное целое
`%x, %o, %b`	Целое в шестнадцатеричном, восьмеричном и двоичном представлениях
`%f, %g, %e`	Числа с плавающей точкой: `3.141593 3.141592653589793 3.141593e+00`
`%t`	Булево значение: `true` или `false`
`%c`	Руна (символ Unicode)
`%s`	Строка
`%q`	Выводит в кавычках строку типа `"abc"` или символ типа `'c'`
`%v`	Любое значение в естественном формате
`%T`	Тип любого значения
`%%`	Символ процента (не требует операнда)

Строка формата в `dup1` содержит также символы табуляции `\t` и новой строки `\n`. Строковые литералы могут содержать такие *управляющие последовательности* для представления символов, которые обычно невидимы на экране и не могут быть введены непосредственно. По умолчанию `Printf` не записывает символ новой строки. По соглашению функции форматирования, имена которых заканчиваются на `f`, такие как `log.Printf` и `fmt.Errorf`, используют правила форматирования `fmt.Printf`, тогда как функции, имена которых заканчиваются на `ln`, как `Println`, форматируют свои аргументы так, как будто используется символ преобразования `%v`, а за ним — символ новой строки.

Многие программы считывают входные данные либо из стандартного ввода, как приведенная выше, либо из последовательности именованных файлов. Следующая версия `dup` может как выполнять чтение стандартного ввода, так и работать со списком файлов, используя `os.Open` для их открывания:

gopl.io/ch1/dup2

```
// Dup2 выводит текст каждой строки, которая появляется во
// входных данных более одного раза. Программа читает
```

```
// стандартный ввод или список именованных файлов.
package main

import (
    "bufio"
    "fmt"
    "os"
)

func main() {
    counts := make(map[string]int)
    files := os.Args[1:]
    if len(files) == 0 {
        countLines(os.Stdin, counts)
    } else {
        for _, arg := range files {
            f, err := os.Open(arg)
            if err != nil {
                fmt.Fprintf(os.Stderr, "dup2: %v\n", err)
                continue
            }
            countLines(f, counts)
            f.Close()
        }
    }
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}

func countLines(f *os.File, counts map[string]int) {
    input := bufio.NewScanner(f)
    for input.Scan() {
        counts[input.Text()]++
    }
    // Примечание: игнорируем потенциальные
    // ошибки из input.Err()
}
```

Функция `os.Open` возвращает два значения. Первое из них является открытым файлом (`*os.File`), который в дальнейшем читает `Scanner`.

Второй результат `os.Open` — значение встроенного типа `error`. Если `err` равно специальному встроенному значению `nil`, файл открыт успешно. Этот файл читается, и по достижении его конца функция `Close` закрывает его и освобождает все связанные с ним ресурсы. С другой стороны, если значение `err` не равно `nil`, значит,

что-то пошло не так. В этом случае значение ошибки описывает происшедшую неприятность. Наша простейшая обработка ошибок выводит сообщение в стандартный поток сообщения об ошибках с помощью вызова `Fprintf` и символов преобразования `%v` (которые выводят значение любого типа в формате по умолчанию), после чего `dup` переходит к следующему файлу; инструкция `continue` выполняет немедленный переход к очередной итерации охватывающего цикла `for`.

Чтобы размеры примеров в книге оставались в разумных пределах, в рассмотренных примерах обработка ошибок отсутствует. Очевидно, мы должны проверить ошибки `os.Open`; однако мы игнорируем менее вероятную возможность ошибки при чтении файла с помощью `input.Scan`. Мы отмечаем места, где пропущена обработка ошибок, а подробнее о ней мы поговорим в разделе 5.4.

Обратите внимание, что вызов `countLines` предшествует объявлению этой функции. Функции и другие объекты уровня пакета могут быть объявлены в любом порядке.

Отображение является *ссылкой* на структуру данных, созданную функцией `make`. Когда отображение передается в функцию, функция получает копию ссылки, поэтому любые изменения, которые вызываемая функция вносит в указываемую структуру данных, будут видны и с помощью исходной ссылки в вызывающем коде. В нашем примере значения, вставляемые в отображение `counts` функцией `countLines`, видны функции `main`.

Рассмотренные выше версии `dup` работают в "потоковом" режиме, в котором входные данные считываются и разбиваются на строки по необходимости, так что в принципе эти программы могут обрабатывать произвольное количество входных данных. Альтернативный подход состоит в чтении всех входных данных в память одним большим "глотком", полном разделении его на строки, и последующей обработке строк. Следующая версия, `dup3`, работает именно таким образом. В ней вводятся функция `ReadFile` (из пакета `io/ioutil`), которая считывает все содержимое именованного файла, и функция `strings.Split`, которая разбивает строку на срез подстрок. (`Split` является противоположностью функции `strings.Join`, с который мы познакомились выше.)

Мы несколько упростили `dup3`. Во-первых, этот вариант программы читает только именованные файлы, но не стандартный ввод, так как функции `ReadFile` требуется аргумент, который представляет собой имя файла. Во-вторых, мы перенесли подсчет строк обратно в функцию `main`, так как теперь он необходим только в одном месте.

gopl.io/ch1/dup3

```
package main

import (
    "fmt"
    "io/ioutil"
    "os"
    "strings"
)
```

```
func main() {
    counts := make(map[string]int)
    for _, filename := range os.Args[1:] {
        data, err := ioutil.ReadFile(filename)
        if err != nil {
            fmt.Fprintf(os.Stderr, "dup3: %v\n", err)
            continue
        }
        for _, line := range strings.Split(string(data), "\n") {
            counts[line]++
        }
    }
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}
```

Функция `ReadFile` возвращает байтовый срез, который должен быть преобразован в `string` так, чтобы его можно было разбить с помощью функции `strings.Split`. Строки и байтовые срезы мы рассмотрим в разделе 3.5.4.

Внутри функции `bufio.Scanner`, `ioutil.ReadFile` и `ioutil.WriteFile` используют методы `Read` и `Write` объекта `*os.File`, но большинству программистов очень редко приходится прибегать к таким низкоуровневым функциям непосредственно. Проще использовать функции более высокого уровня наподобие функций из пакетов `bufio` и `io/ioutil`.

Упражнение 1.4. Измените программу `dup2` так, чтобы она выводила имена всех файлов, в которых найдены повторяющиеся строки.

1.4. Анимированные GIF-изображения

Следующая программа демонстрирует основы применения стандартных пакетов Go для работы с изображениями, возможности которых мы используем для создания последовательности растровых изображений и их сборки в анимированное GIF-изображение. Эти изображения, именуемые *фигурами Лиссажу*, часто использовались в качестве визуальных эффектов в фантастических фильмах 1960-х годов. Это параметрические кривые, полученные с помощью гармонических колебаний в двух измерениях, таких как две синусоиды, поданные на входы x и y осциллографа. На рис. 1.1 приведено несколько примеров таких фигур.

В коде этой программы есть несколько новых конструкций, включая `const`-объявления, структурные типы и составные литералы. В отличие от большинства наших примеров здесь также имеются вычисления с плавающей точкой. Подробно эти темы рассматриваются в следующих главах, а сейчас наша основная цель — дать вам

представление о том, как выглядит Go и что можно легко сделать с помощью этого языка программирования и его библиотек.

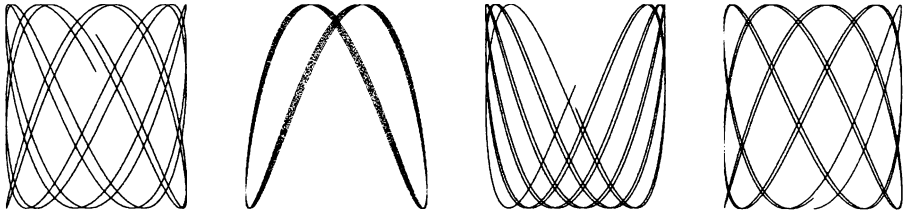


Рис. 1.1. Четыре фигуры Лиссажу

gopl.io/ch1/lissajous

```
// Lissajous генерирует анимированный GIF из случайных
// фигур Лиссажу.
package main

import (
    "image"
    "image/color"
    "image/gif"
    "io"
    "math"
    "math/rand"
    "os"
)

var palette = []color.Color{color.White, color.Black}

const (
    whiteIndex = 0 // Первый цвет палитры
    blackIndex = 1 // Следующий цвет палитры
)

func main() {
    lissajous(os.Stdout)
}
func lissajous(out io.Writer) {
    const (
        cycles  = 5     // Количество полных колебаний x
        res     = 0.001 // Угловое разрешение
        size    = 100   // Канва изображения охватывает [size..+size]
        nframes = 64    // Количество кадров анимации
        delay   = 8     // Задержка между кадрами (единица - 10мс)
    )
    rand.Seed(time.Now().UTC().UnixNano())
    freq := rand.Float64() * 3.0 // Относительная частота колебаний y
    anim := gif.GIF{LoopCount: nframes}
```

```
    phase := 0.0                // Разность фаз
    for i := 0; i < nframes; i++ {
        rect := image.Rect(0, 0, 2*size+1, 2*size+1)
        img := image.NewPaletted(rect, palette)
        for t := 0.0; t < cycles*2*math.Pi; t += res {
            x := math.Sin(t)
            y := math.Sin(t*freq + phase)
            img.SetColorIndex(size+int(x*size+0.5), size+int(y*size+0.5),
                              blackIndex)
        }
        phase += 0.1
        anim.Delay = append(anim.Delay, delay)
        anim.Image = append(anim.Image, img)
    }
    gif.EncodeAll(out, &anim)     // Примечание: игнорируем ошибки
}
```

После импорта пакета, путь к которому содержит несколько компонент (наподобие `image/color`), мы обращаемся к пакету по имени последнего компонента. Таким образом, переменная `color.White` принадлежит пакету `image/color`, а `gif.GIF` принадлежит пакету `image/gif`.

Объявление `const` (§3.6) дает имена константам, т.е. значениям, которые фиксированы во время компиляции, таким как числовые параметры циклов, задержки и т.п. Подобно объявлениям `var`, объявления `const` могут находиться на уровне пакетов (так что имена видимы везде в пакете) или в функции (так что имена видимы только в пределах данной функции). Значение константы должно быть числом, строкой или логическим значением.

Выражения `[]color.Color{...}` и `gif.GIF{...}` являются *составными литералами* (composite literals; разделы 4.2, 4.4.1) — компактной записью для инстанцирования составных типов Go из последовательности значений элементов. В приведенном примере первый из них представляет собой срез, а второй — *структуру*.

Тип `gif.GIF` является структурным типом (см. раздел 4.4). Структура представляет собой группу значений, именуемых *полями*, зачастую различных типов, которые собраны в один объект, рассматриваемый как единое целое. Переменная `anim` является структурой типа `gif.GIF`. Структурный литерал создает значение структуры, поле `LoopCount` которого устанавливается равным `nframes`; все прочие поля имеют нулевые значения соответствующих типов. Обращение к отдельным полям структуры выполняется с помощью записи с точкой, как в двух последних присваиваниях, которые явно обновляют поля `Delay` и `Image` переменной `anim`.

Функция `lissajous` содержит два вложенных цикла. Внешний цикл выполняет 64 итерации, каждая из которых генерирует отдельный кадр анимации. Она создает новое изображение размером 201×201 с палитрой из двух цветов, белого и черного. Все пиксели изначально устанавливаются равными нулевому значению палитры, т.е. имеют нулевой цвет, который мы определили как белый. Каждая итерация внешнего цикла генерирует новое изображение путем установки черного цвета для некоторых пикселей. Результат добавляется (с помощью встроенной функции `append`

(раздел 4.2.1)) к списку кадров в `anim`, вместе с указанной ранее задержкой в 80 мс. Наконец последовательность кадров и задержек кодируется в виде изображения в формате GIF и записывается в выходной поток `out`. Типом `out` является `io.Writer`, что позволяет выполнять запись в множество возможных целевых мест назначения, как мы вскоре покажем.

Внутренний цикл запускает два осциллятора. Осциллятор *x* представляет собой простую функцию синуса. Осциллятор *y* также является синусоидой, но ее частота относительно частоты осциллятора *x* является случайным числом от 0 до 3, а его фаза относительно осциллятора *x* изначально равна 0, но увеличивается с каждым кадром анимации. Цикл выполняется до тех пор, пока осциллятор *x* не выполнит пять полных циклов. На каждом шаге вызывается функция `SetColorIndex` для того, чтобы окрасить пиксель, соответствующий координате (*x*,*y*), в черный цвет (позиция 1 в палитре).

Функция `main` вызывает функцию `lissajous`, заставляя ее осуществлять запись в стандартный вывод, так что приведенная далее команда генерирует анимированный GIF с кадрами, подобными показанным на рис. 1.1.

```
$ go build gopl.io/ch1/lissajous
$ ./lissajous >out.gif
```

Упражнение 1.5. Измените палитру программы `lissajous` так, чтобы изображение было зеленого цвета на черном фоне, чтобы быть более похожим на экран осциллографа. Чтобы создать веб-цвет `#RRGGBB`, воспользуйтесь инструкцией `color.RGBA{0xRR,0xGG,0xBB,0xff}`, в которой каждая пара шестнадцатеричных цифр представляет яркость красного, зеленого и синего компонентов пикселя.

Упражнение 1.6. Измените программу `lissajous` так, чтобы она генерировала изображения разных цветов, добавляя в палитру `palette` больше значений, а затем выводя их путем изменения третьего аргумента функции `SetColorIndex` некоторым нетривиальным способом.

1.5. Выборка URL

Для множества приложений доступ к информации в Интернете не менее важен, чем доступ к локальной файловой системе. Go предоставляет коллекцию пакетов, сгруппированную в каталоге `net`, которая облегчает отправку и получение информации из Интернета, создание низкоуровневых соединений и настройку серверов, для чего в особенности полезны возможности параллелизации Go (вводимые в главе 8, "Go-подпрограммы и каналы").

Для иллюстрации минимально необходимого кода для получения информации по протоколу HTTP ниже приведена простая программа `fetch`, которая осуществляет выборку содержимого по каждому из указанных URL и выводит его как не интерпретированный текст; эта программа создана по мотивам неоценимо полезной утилиты `curl`. Понятно, что обычно с такими данными должны выполняться некоторые дей-

ствия, но здесь мы просто демонстрируем основную идею. Мы часто будем использовать эту программу далее в книге.

gopl.io/ch1/fetch

```
// Fetch выводит ответ на запрос по заданному URL.
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "os"
)

func main() {
    for _, url := range os.Args[1:] {
        resp, err := http.Get(url)
        if err != nil {
            fmt.Fprintf(os.Stderr, "fetch: %v\n", err)
            os.Exit(1)
        }
        b, err := ioutil.ReadAll(resp.Body)
        resp.Body.Close()
        if err != nil {
            fmt.Fprintf(os.Stderr, "fetch: чтение %s: %v\n", url, err)
            os.Exit(1)
        }
        fmt.Printf("%s", b)
    }
}
```

В этой программе вводятся функции из двух пакетов — `net/http` и `io/ioutil`. Функция `http.Get` выполняет HTTP-запрос и при отсутствии ошибок возвращает результат в структуре `resp`. Поле `Body` этой структуры содержит ответ сервера в виде потока, доступного для чтения. Затем `ioutil.ReadAll` считывает весь ответ; результат сохраняется в переменной `b`. Поток `Body` закрывается для предотвращения утечки ресурсов, и функция `Printf` записывает ответ в стандартный вывод.

```
$ go build gopl.io/ch1/fetch
$ ./fetch http://gopl.io
<html>
<head>
<title>The Go Programming Language</title>
...
```

В случае ошибки HTTP-запроса `fetch` сообщает о том, что произошло:

```
$ ./fetch http://bad.gopl.io
fetch: Get http://bad.gopl.io: dial tcp: lookup bad.gopl.io: no such host
```

В случае любой ошибки `os.Exit(1)` завершает работу процесса с кодом состояния 1.

Упражнение 1.7. Вызов функции `io.Copy(dst,src)` выполняет чтение `src` и запись в `dst`. Воспользуйтесь ею вместо `ioutil.ReadAll` для копирования тела ответа в поток `os.Stdout` без необходимости выделения достаточно большого для хранения всего ответа буфера. Не забудьте проверить, не произошла ли ошибка при вызове `io.Copy`.

Упражнение 1.8. Измените программу `fetch` так, чтобы к каждому аргументу URL автоматически добавлялся префикс `http://` в случае отсутствия в нем такового. Можете воспользоваться функцией `strings.HasPrefix`.

Упражнение 1.9. Измените программу `fetch` так, чтобы она выводила код состояния HTTP, содержащийся в `resp.Status`.

1.6. Параллельная выборка URL

Одним из наиболее интересных и новых аспектов Go является поддержка параллельного программирования. Это большая тема, которой посвящены главы 8, "Go-подпрограммы и каналы", и 9, "Параллельность и совместно используемые переменные", так что сейчас мы просто дадим вам возможность попробовать на вкус основные механизмы параллельности Go — go-подпрограммы и каналы.

Следующая программа, `fetchall`, точно так же выполняет выборку содержимого URL, как и в предыдущем примере, но делает это по многим URL одновременно, так что этот процесс займет не больше времени, чем самая долгая выборка (а не время, составляющее сумму всех времен отдельных выборок). Эта версия `fetchall` игнорирует ответы серверов, но сообщает об их размерах и затраченном на их получение времени:

gopl.io/ch1/fetchall

```
// Fetchall выполняет параллельную выборку URL и сообщает
// о затраченном времени и размере ответа для каждого из них.
package main

import (
    "fmt"
    "io"
    "io/ioutil"
    "net/http"
    "os"
    "time"
)

func main() {
    start := time.Now()
    ch := make(chan string)
    for _, url := range os.Args[1:] {
```

```
        go fetch(url, ch) // Запуск go-подпрограммы
    }
    for range os.Args[1:] {
        fmt.Println(<-ch)
        // Получение из канала ch
    }
    fmt.Printf("%.2fs elapsed\n", time.Since(start).Seconds())
}
func fetch(url string, ch chan<- string) {
    start := time.Now()
    resp, err := http.Get(url)
    if err != nil {
        ch <- fmt.Sprint(err) // Отправка в канал ch
        return
    }
    nbytes, err := io.Copy(ioutil.Discard, resp.Body)
    resp.Body.Close() // Исключение утечки ресурсов
    if err != nil {
        ch <- fmt.Sprintf("while reading %s: %v", url, err)
        return
    }
    secs := time.Since(start).Seconds()
    ch <- fmt.Sprintf("%.2fs %7d %s", secs, nbytes, url)
}
```

Вот пример работы программы:

```
$ go build gopl.io/ch1/fetchall
$ ./fetchall https://golang.org http://gopl.io https://godoc.org
0.14s 6852 https://godoc.org
0.16s 7261 https://golang.org
0.48s 2475 http://gopl.io
0.48s elapsed
```

go-подпрограмма представляет собой параллельное выполнение функции. *Канал* является механизмом связи, который позволяет одной go-подпрограмме передавать значения определенного типа другой go-подпрограмме. Функция `main` выполняется в go-подпрограмме, а инструкция `go` создает дополнительные go-подпрограммы.

Функция `main` создает канал строк с помощью `make`. Для каждого аргумента командной строки инструкция `go` в первом цикле по диапазону запускает новую go-подпрограмму, которую `fetch` вызывает асинхронно для выборки URL с помощью `http.Get`. Функция `io.Copy` считывает тело ответа и игнорирует его, записывая в выходной поток `ioutil.Discard`. `Copy` возвращает количество байтов и информацию о происшедших ошибках. При получении каждого результата `fetch` отправляет итоговую строку в канал `ch`. Второй цикл по диапазону в функции `main` получает и выводит эти строки.

Когда одна go-подпрограмма пытается отправить или получить информацию по каналу, она блокируется, пока другая go-подпрограмма пытается выполнить соот-

ветствующие операции получения или отправки, и после передачи информации обе go-подпрограммы продолжают работу. В данном примере каждая функция `fetch` отправляет значение (`ch <-` *expression*) в канал `ch`, и `main` получает их все (`<- ch`). То, что весь вывод осуществляется функцией `main`, гарантирует, что вывод каждой go-подпрограммы будет обработан как единое целое, без опасности получить на экране чередование вывода при завершении двух go-подпрограмм в один и тот же момент времени.

Упражнение 1.10. Найдите веб-сайт, который содержит большое количество данных. Исследуйте работу кеширования путем двукратного запуска `fetchall` и сравнения времени запросов. Получаете ли вы каждый раз одно и то же содержимое? Измените `fetchall` так, чтобы вывод осуществлялся в файл и чтобы затем можно было его изучить.

Упражнение 1.11. Выполните `fetchall` с длинным списком аргументов, таким как образцы, доступные на сайте `alexa.com`. Как ведет себя программа, когда веб-сайт просто не отвечает? (В разделе 8.9 описан механизм отслеживания таких ситуаций.)

1.7. Веб-сервер

Библиотека Go делает написание веб-сервера, который отвечает на запросы клиентов наподобие осуществляемых программой `fetch`, простым и легким. В этом разделе мы покажем минимальный сервер, который возвращает компонент пути из URL, использованного для обращения к серверу. Иначе говоря, если запрос имеет вид `http://localhost:8000/hello`, то ответ выглядит как `URL.Path = "/hello"`.

gopl.io/ch1/server1

```
// Server1 — минимальный "echo"-сервер.
package main

import (
    "fmt"
    "log"
    "net/http"
)

func main() {
    http.HandleFunc("/", handler) // Каждый запрос вызывает обработчик
    log.Fatal(http.ListenAndServe("localhost:8000", nil))
}

// Обработчик возвращает компонент пути из URL запроса.
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "URL.Path = %q\n", r.URL.Path)
}
```

Программа содержит буквально несколько строк, потому что библиотечные функции выполняют бо́льшую часть работы. Функция `main` связывает функцию-обработчик с входящим URL, который начинается с `/`, и запускает сервер, прослушивающий порт 8000 в ожидании входящих запросов. Запрос представлен структурой типа `http.Request`, которая содержит ряд связанных полей, одно из которых представляет собой URL входящего запроса. Полученный запрос передается функции-обработчику, которая извлекает компонент пути (`/hello`) из URL запроса и отправляет его обратно в качестве ответа с помощью `fmt.Fprintf`. Веб-серверы подробно рассмотрены в разделе 7.7.

Давайте запустим сервер в фоновом режиме. В Mac OS X или Linux добавьте к команде амперсанд (&); в Microsoft Windows необходимо запустить команду без амперсанда в отдельном окне.

```
$ go run src/gopl.io/ch1/server1/main.go &
```

Затем осуществим клиентский запрос из командной строки:

```
$ go build gopl.io/ch1/fetch
$ ./fetch http://localhost:8000
URL.Path = "/"
$ ./fetch http://localhost:8000/help
URL.Path = "/help"
```

Можно также обратиться к серверу с помощью веб-браузера, как показано на рис. 1.2.

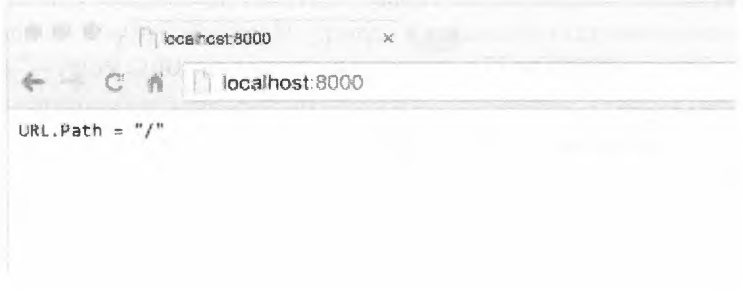


Рис. 1.2. Ответ от сервера

Расширять возможности сервера довольно легко. Одним полезным расширением является конкретный URL, который возвращает некоторое состояние. Например, эта версия делает то же, что и предыдущая, но при этом еще и подсчитывает количество запросов; запрос к URL `/count` возвращает это количество, за исключением самого запроса `/count`:

gopl.io/ch1/server2

```
// Server2 — минимальный "echo"-сервер со счетчиком запросов.
package main

import (
    "fmt"
```

```
    "log"
    "net/http"
    "sync"
)

var mu sync.Mutex
var count int

func main() {
    http.HandleFunc("/", handler)
    http.HandleFunc("/count", counter)
    log.Fatal(http.ListenAndServe("localhost:8000", nil))
}

// Обработчик, возвращающий компонент пути запрашиваемого URL.
func handler(w http.ResponseWriter, r *http.Request) {
    mu.Lock()
    count++
    mu.Unlock()
    fmt.Fprintf(w, "URL.Path = %q\n", r.URL.Path)
}

// Счетчик, возвращающий количество сделанных запросов.
func counter(w http.ResponseWriter, r *http.Request) {
    mu.Lock()
    fmt.Fprintf(w, "Count %d\n", count)
    mu.Unlock()
}
```

У сервера имеется два обработчика, и запрашиваемый URL определяет, какой из них будет вызван: запрос `/count` вызывает `counter`, а все прочие — `handler`. Сервер запускает обработчик для каждого входящего запроса в отдельной go-подпрограмме, так что несколько запросов могут обрабатываться одновременно. Однако если два параллельных запроса попытаются обновить счетчик `count` в один и тот же момент времени, он может быть увеличен не согласованно; в такой программе может возникнуть серьезная ошибка под названием *состояние гонки* (race condition; см. раздел 9.1). Чтобы избежать этой проблемы, нужно гарантировать, что доступ к переменной получает не более одной go-подпрограммы одновременно. Для этого каждый доступ к переменной должен быть окружен вызовами `mu.Lock()` и `mu.Unlock()`. Более детально параллельность с совместно используемыми переменными будет рассматриваться в главе 9, "Параллельность и совместно используемые переменные".

В качестве примера, более богатого функциональностью, функция обработчика может сообщать о заголовках и данных, которые она получает, тем самым создавая полезный для проверки и отладки запросов сервер:

gopl.io/ch1/server3

```
// Обработчик HTTP-запросов.
func handler(w http.ResponseWriter, r *http.Request) {
```

```
    fmt.Fprintf(w, "%s %s %s\n", r.Method, r.URL, r.Proto)
    for k, v := range r.Header {
        fmt.Fprintf(w, "Header[%q] = %q\n", k, v)
    }
    fmt.Fprintf(w, "Host = %q\n", r.Host)
    fmt.Fprintf(w, "RemoteAddr = %q\n", r.RemoteAddr)
    if err := r.ParseForm(); err != nil {
        log.Print(err)
    }
    for k, v := range r.Form {
        fmt.Fprintf(w, "Form[%q] = %q\n", k, v)
    }
}
```

Здесь используются поля структуры `http.Request` для генерации вывода наподобие следующего:

```
GET /?q=query HTTP/1.1
Header["AcceptEncoding"] = ["gzip, deflate, sdch"]
Header["AcceptLanguage"] = ["enUS, en;q=0.8"]
Header["Connection"] = ["keepalive"]
Header["Accept"] =
↳         ["text/html,application/xhtml+xml,application/xml;..."]
Header["UserAgent"] =
↳         ["Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_5)..."]
Host = "localhost:8000"
RemoteAddr = "127.0.0.1:59911"
Form["q"] = ["query"]
```

Обратите внимание на то, как вызов `ParseForm` оказывается вложенным в конструкцию `if`. Go разрешает простым инструкциям, таким как объявление локальной переменной, предшествовать условию `if`, что особенно полезно при обработке ошибок, как в приведенном примере. Мы могли бы записать это следующим образом:

```
err := r.ParseForm()
if err != nil {
    log.Print(err)
}
```

Однако объединение инструкций оказывается более коротким и уменьшает область видимости переменной `err`, что является хорошей практикой. Области видимости будут рассматриваться в разделе 2.7.

В этих программах мы видели применение в качестве выходных потоков трех очень разных типов. Программа `fetch` копирует ответ HTTP в `os.Stdout`, в файл, как делала программа `lissajous`. Программа `fetchall` отбрасывает ответ (но вычисляет при этом его длину), копируя его в `ioutil.Discard`. А приведенный выше веб-сервер использует `fmt.Fprintf` для записи в `http.ResponseWriter`, который является представлением веб-браузера.

Хотя эти три типа различаются деталями работы, все они удовлетворяют общему *интерфейсу* и позволяют использовать любой из них там, где требуется выходной поток. Этот интерфейс — `io.Writer` — подробнее рассматривается в разделе 7.1.

Механизм интерфейса Go является темой главы 7, "Интерфейсы", но, чтобы иметь представление о нем и о том, на что он способен, давайте посмотрим, как можно легко объединить веб-сервер с функцией `lissajous` так, чтобы анимированные изображения выводились не в стандартный вывод, а в клиент HTTP. Просто добавьте эти строки к веб-серверу:

```
handler := func(w http.ResponseWriter, r *http.Request) {
    lissajous(w)
}
http.HandleFunc("/", handler)
```

или, что то же самое:

```
http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    lissajous(w)
})
```

Вторым аргументом функции `HandleFunc` является *литерал функции*, т.е. анонимная функция, определенная в точке использования. Дальнейшие пояснения по этому вопросу вы найдете в разделе 5.6.

Если вы внесли указанное изменение, посетите адрес `http://localhost:8000` с помощью вашего браузера. Всякий раз при загрузке страницы вы увидите анимацию наподобие приведенной на рис. 1.3.

Упражнение 1.12. Измените сервер с фигурами Лиссажу так, чтобы значения параметров считывались из URL. Например, URL вида `http://localhost:8000/?cycles=20` устанавливает количество циклов равным 20 вместо значения по умолчанию, равного 5. Используйте функцию `strconv.Atoi` для преобразования строкового параметра в целое число. Просмотреть документацию по данной функции можно с помощью команды `go doc strconv.Atoi`.

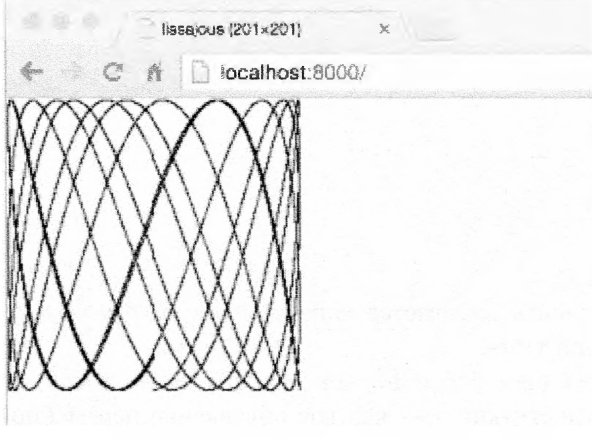


Рис. 1.3. Анимированные фигуры Лиссажу в браузере

1.8. Некоторые мелочи

В Go есть еще очень много такого, о чем не рассказано в этом кратком введении. Далее перечислены некоторые из тем, которых мы едва коснулись или вовсе не касались. Здесь приводится очень немного информации просто для того, чтобы вы знали, с чем имеете дело, сталкиваясь с таким кодом до того, как эти возможности будут рассмотрены подробнее.

Управление потоком. Мы рассмотрели две фундаментальные инструкции управления потоком, `if` и `for`, но не инструкцию `switch`, которая представляет собой инструкцию множественного ветвления. Вот небольшой пример такой инструкции:

```
switch coinflip() {
case "heads":
    heads++
case "tails":
    tails++
default:
    fmt.Println("Приземлились!")
}
```

Результат вызова `coinflip` сравнивается со значением в каждой части `case`. Значения проверяются сверху вниз, и при первом найденном совпадении выполняется соответствующий код. Необязательный вариант `default` выполняется, если нет совпадения ни с одним из перечисленных значений; он может находиться где угодно. "Проваливание" из одного `case` в другой, как это происходит в C-подобных языках, в Go отсутствует (хотя в языке имеется редко используемая инструкция `fallthrough`, переопределяющая это поведение).

Инструкция `switch` может обойтись и без операнда; она может просто перечислять различные инструкции `case`, каждая из которых при этом представляет собой логическое выражение:

```
func Signum(x int) int {
    switch {
    case x > 0:
        return +1
    default:
        return 0
    case x < 0:
        return 1
    }
}
```

Такая разновидность называется *переключателем без тегов*; она эквивалентна конструкции `switch true`.

Подобно инструкциям `for` и `if`, конструкция `switch` может включать необязательную простую инструкцию — краткое объявление переменной, инструкцию ин-

кремента или присваивания или вызов функции, который может задавать тестируемое значение.

Инструкции `break` и `continue` модифицируют поток управления. Инструкция `break` заставляет передать управление следующей инструкции после наиболее глубоко вложенной инструкции `for`, `switch` или `select` (с ней мы познакомимся позже), и, как мы видели в разделе 1.3, инструкция `continue` заставляет наиболее глубоко вложенный цикл `for` начать очередную итерацию. Инструкции могут иметь метки, так что `break` и `continue` могут на них ссылаться, например для одновременного прекращения работы нескольких вложенных циклов или для начала очередной итерации внешнего цикла. Имеется даже инструкция `goto`, хотя она предназначена для машинно-генерируемого кода, а не для использования программистами.

Именованные типы. Объявление `type` позволяет присваивать имена существующим типам. Поскольку структурные типы зачастую длинны, они почти всегда именованы. Простейшим примером является определение типа `Point` для двумерной графики:

```
type Point struct {
    X, Y int
}
var p Point
```

Объявления типов и именованные типы рассматриваются в главе 2, "Структура программы".

Указатели. Go предоставляет указатели, т.е. значения, содержащие адреса переменных. В одних языках, в особенности в C, указатели являются относительно неограниченными. В других языках программирования указатели маскируются под "ссылки", и с ними можно сделать не так уж много операций, кроме их передачи в функции. Go занимает промежуточную позицию. Указатели в нем являются явно видимыми. Оператор & дает адрес переменной, а оператор * позволяет получить значение переменной, на которую указывает указатель; однако арифметики указателей в Go нет. Указатели будут подробнее рассматриваться в разделе 2.3.2.

Методы и интерфейсы. Метод представляет собой функцию, связанную с именованным типом; Go необычен тем, что методы могут быть связаны почти с любым именованным типом. Методы рассматриваются в главе 6, "Методы". Интерфейсы представляют собой абстрактные типы, которые позволяют рассматривать различные конкретные типы одинаково, на основе имеющихся у них методов, без учета того, как они представлены или реализованы. Интерфейсам посвящена глава 7, "Интерфейсы".

Пакеты. Go поставляется с обширной стандартной библиотекой полезных пакетов; кроме того, еще множество пакетов создано и распространено сообществом пользователей Go. Программирование чаще состоит в использовании существующих пакетов, чем в написании исходного кода собственных пакетов. В книге мы будет иметь дело с парой десятков самых важных стандартных пакетов, но на самом деле их намного больше, чем у нас есть места для их упоминания. Мы не можем предоста-

вить вам что-либо даже отдаленно напоминающее справочник по пакетам, как бы мы этого ни хотели.

Прежде чем начать работу над любой новой программой, стоит посмотреть, какие существующие пакеты могут облегчить выполнение работы. Вы можете найти предметный указатель стандартной библиотеки по адресу `https://golang.org/pkg`, а пакеты, предоставленные сообществом, — по адресу `https://godoc.org`. Инструмент `go doc` делает эти документы легко доступными из командной строки:

```
$ go doc http.ListenAndServe
package http // import "net/http"

func ListenAndServe(addr string, handler Handler) error

    ListenAndServe прослушивает сетевой адрес TCP и...
```

Комментарии. Мы уже упоминали о комментарии в начале исходного текста, документирующем программу или пакет. Хороший стиль требует написания комментария перед объявлением каждой функции, описывающим ее поведение. Это важные соглашения, поскольку они используются такими инструментами, как `go doc` и `godoc`, для поиска и отображения документации (раздел 10.7.4).

Для комментариев, которые охватывают несколько строк или появляются в выражении или операторе, можно использовать запись `/ *... */`, знакомую по другим языкам. Такие комментарии иногда используются в начале файла для большого блока пояснительного текста, чтобы избежать символов `//` в начале каждой строки. В пределах комментария символы `//` и `/*` специального значения не имеют, так что не пытайтесь создавать вложенные комментарии.

2

Структура программы

В Go, как и в любом другом языке программирования, большие программы строятся из небольшого набора базовых конструкций. Переменные хранят значения. Простые выражения объединяются в более сложные с помощью операций, таких как сложение и вычитание. Базовые типы собираются в агрегаты, такие как массивы и структуры. Выражения используются в инструкциях, порядок выполнения которых определяется инструкциями управления потоком выполнения, такими как `if` или `for`. Инструкции сгруппированы в функции для изоляции от прочего кода и повторного использования. Функции собираются в исходные файлы и пакеты.

Мы видели большинство приведенных примеров в предыдущей главе. В этой главе мы более подробно поговорим об основных структурных элементах программы Go. Примеры программ намеренно просты, так что мы можем сосредоточиться на языке без необходимости прибегать к сложным алгоритмам и структурам данных.

2.1. Имена

Имена функций, переменных, констант, типов, меток инструкций и пакетов в Go следуют простому правилу: имя начинается с буквы (т.е. со всего, что Unicode считает буквой) или с подчеркивания и может иметь сколько угодно дополнительных букв, цифр и подчеркиваний. Имена чувствительны к регистру: `heapSort` и `Heapsort` являются разными именами.

Go имеет 25 *ключевых слов* наподобие `if` и `switch`, которые могут использоваться только там, где разрешает синтаксис языка программирования; они не могут использоваться в качестве имен.

```
break       default      func        interface   select
case        defer        go          map         struct
chan        else         goto        package     switch
const       fallthrough  if          range       type
continue    for          import      return      var
```

Кроме того, имеется около трех десятков *предопределенных* имен наподобие `int` и `true` для встроенных констант, типов и функций.

Константы:	`true false iota nil`

Типы:	`int int8 int16 int32 int64`
	`uint uint8 uint16 uint32 uint64 uintptr`
	`float32 float64 complex128 complex64`
	`bool byte rune string error`

Функции:	`make len cap new append copy close delete`
	`complex real imag`
	`panic recover`

Эти имена не являются зарезервированными, так что можете использовать их в объявлениях. Мы увидим несколько мест, где переобъявление одного из них имеет смысл, но остерегайтесь возможной путаницы!

Если некая сущность объявлена внутри функции, она является *локальной* для данной функции. Если сущность объявлена вне функции, она видна во всех файлах пакета, к которому она принадлежит. Регистр первой буквы имени определяет его видимость через границы пакета. Если имя начинается с прописной буквы, оно *экспортируемое*; это означает, что оно видимо и доступно за пределами собственного пакета и к нему могут обращаться другие части программы, как в случае функции `Printf` из пакета `fmt`. Сами имена пакетов всегда состоят из строчных букв.

Не существует никаких ограничений на длину имени, но соглашения и стиль программирования в Go отдают предпочтение коротким именам, в особенности для локальных переменных с небольшими областями видимости; гораздо больше шансов встретить переменную `i`, чем переменную `theLoopIndex`. Словом, чем больше область видимости имени, тем длиннее и значимее оно должно быть.

Стилистически программисты Go обычно при формировании имен путем сочетания слов используют соединение слов, начинающихся с прописных букв (т.е. использование внутренних заглавных букв предпочтительнее внутренних подчеркиваний). Таким образом, стандартные библиотеки имеют функции с такими именами, как `QuoteRuneToASCII` или `parseRequestLine`, но никогда с `quote_rune_to_ASCII` или `parse_request_line`. Буквы сокращений наподобие `ASCII` и `HTML` всегда отображаются в одном и том же регистре, поэтому функция может быть названа `htmlEscape`, `HTMLEscape` или `escapeHTML`, но не `escapeHtml`.

2.2. Объявления

Объявление именует программную сущность и определяет некоторые (или все) ее свойства. Имеется четыре основные разновидности объявлений: `var`, `const`, `type` и `func`. В этой главе речь пойдет о переменных и типах, в главе 3, "Фундаментальные типы данных" — о константах, а в главе 5, "Функции" — о функциях.

Программа Go хранится в одном или нескольких файлах, имена которых заканчиваются на `.go`. Каждый файл начинается с объявления `package`, которое говорит, частью какого пакета является данный файл. За объявлением `package` следуют любые объявления `import`, а затем — последовательность объявления типов, переменных,

констант и функций *уровня пакета* в произвольном порядке. Например, приведенная ниже программа объявляет константу, функцию и пару переменных:

gopl.io/ch2/boiling

```
// Boiling выводит температуру кипения воды.
package main

import "fmt"

const boilingF = 212.0

func main() {
    var f = boilingF
    var c = (f - 32) * 5 / 9
    fmt.Printf("Температура кипения = %g°F или %g°C\n", f, c)
    // Вывод:
    // Температура кипения = 212°F или 100°C
}
```

Константа `boilingF` представляет собой объявление уровня пакета (как и функция `main`), тогда как переменные `f` и `c` являются локальными для функции `main`. Имя каждой сущности уровня пакета является видимым не только в исходном файле, содержащем ее объявление, но и во всех файлах пакета. Локальные объявления, напротив, видимы только в пределах функции, в которой они объявлены, а возможно, и только в небольшой ее части.

Объявление функции содержит имя, список параметров (переменные, значения которых предоставляет вызывающая функция), необязательный список результатов и тело функции, которое содержит операторы, определяющие, что именно делает функция. Список результатов опускается, если функция ничего не возвращает. Выполнение функции начинается с первого оператора и продолжается до тех пор, пока не встретится оператор `return` или не будет достигнут конец функции, которая не возвращает результаты. Затем управление и результаты возвращаются вызывающей функции.

Мы уже видели немалое количество функций, и нас ждет намного большее их количество, включая детальное обсуждение функций в главе 5, "Функции". Так что все, что мы рассмотрим сейчас, — это не более чем эскиз. Функция `fToC`, показанная ниже, инкапсулирует логику преобразования температуры. Она определена только один раз, но может использоваться в нескольких местах. В приведенном исходном тексте функция `main` вызывает ее дважды, используя значения двух различных локальных констант:

gopl.io/ch2/ftoc

```
// Ftoc выводит результаты преобразования двух температур
// по Фаренгейту в температуру по Цельсию.
package main

import "fmt"
```

```
func main() {
    const freezingF, boilingF = 32.0, 212.0
    fmt.Printf("%g°F = %g°C\n", freezingF, fToC(freezingF))
    // "32°F = 0°C"
    fmt.Printf("%g°F = %g°C\n", boilingF, fToC(boilingF))
    // "212°F = 100°C"
}

func fToC(f float64) float64 {
    return (f - 32) * 5 / 9
}
```

2.3. Переменные

Объявление `var` создает переменную определенного типа, назначает ей имя и присваивает начальное значение. Каждое объявление имеет общий вид

```
var name type = expression
```

Любая из частей `type` и `= expression` может быть опущена, но не обе вместе. Если опущен тип, он определяется из инициализирующего выражения. Если же опущено выражение, то начальным значением является *нулевое значение* для данного типа, равное `0` для чисел, `false` — для булевых переменных, `""` — для строк и `nil` — для интерфейсов и ссылочных типов (срезов, указателей, отображений, каналов, функций). Нулевое значение составного типа наподобие массивов или структур представляет собой нулевые значения всех его элементов или полей.

Механизм нулевого значения гарантирует, что переменные всегда хранят вполне определенные значения своего типа; в Go не существует такого понятия, как неинициализированная переменная. Это упрощает код и зачастую обеспечивает разумное поведение граничных условий без дополнительной работы. Например, код

```
var s string
fmt.Println(s) // ""
```

выводит пустую строку, а не приводит к ошибке или непредсказуемому поведению. Программисты на Go часто прикладывают некоторые дополнительные усилия, чтобы сделать нулевое значение более сложного типа значащим, так что эти переменные начинают свое существование в полезном состоянии.

В одном объявлении можно объявлять и (необязательно) инициализировать несколько переменных, используя соответствующий список выражений. Пропущенный тип позволяет объявлять несколько переменных разных типов.

```
var i, j, k int                 // int, int, int
var b, f, s = true, 2.3, "four" // bool, float64, string
```

Инициализаторы могут быть литеральными значениями или произвольными выражениями. Переменные уровня пакета инициализируются до начала выполнения

функции `main` (раздел 2.6.2), а локальные переменные инициализируются тогда, когда в процессе выполнения функции встречаются их объявления.

Множество переменных может также быть инициализировано с помощью вызова функции, возвращающей несколько значений:

```
var f, err = os.Open(name) // os.Open возвращает файл и ошибку
```

2.3.1. Краткое объявление переменной

В функции для объявления и инициализации локальных переменных может использоваться альтернативная форма объявления, именуемая *кратким объявлением переменной*. Она имеет вид `name := expression`, и тип переменной `name` определяется как тип выражения `expression`. Вот три из множества кратких объявлений переменных в функции `lissajous` (раздел 1.4):

```
anim := gif.GIF{LoopCount: nframes}
freq := rand.Float64() * 3.0
t := 0.0
```

В силу краткости и гибкости краткие объявления переменных используются для объявления и инициализации большинства локальных переменных. Объявление `var`, как правило, резервируется для локальных переменных, требующих явного указания типа, который отличается от выражения инициализатора, или когда значение переменной будет присвоено позже, а его начальное значение не играет роли.

```
i := 100                  // int
var boiling float64 = 100 // float64
var names []string
var err error
var p Point
```

Как и в случае объявления `var`, в одном кратком объявлении можно объявить и инициализировать несколько переменных, например

```
i, j := 0, 1
```

Однако объявления с несколькими выражениями инициализации следует использовать только тогда, когда они могут повысить удобочитаемость, как, например, в коротких и естественных группах наподобие инициализирующей части цикла `for`.

Не забывайте, что объявление — это `:=`, а `=` — это присваивание. Объявление нескольких переменных не следует путать с *присваиванием кортежу* (раздел 2.4.1), в котором каждой переменной в левой части инструкции присваивается значение из правой части:

```
i, j = j, i // Обмен значений i и j
```

Подобно обычным объявлениям `var`, краткие объявления переменных могут использоваться для вызовов функций наподобие `os.Open`, которые возвращают два или больше значений:

```
f, err := os.Open(name)
if err != nil {
    return err
}
// ... использование f ...
f.Close()
```

Один тонкий, но важный момент: краткое объявление переменной не обязательно *объявляет* все переменные в своей левой части. Если некоторые из них уже были объявлены в *том же* лексическом блоке (раздел 2.7), то для этих переменных краткие объявления действуют как *присваивания*.

В приведенном ниже коде первая инструкция объявляет как `in`, так и `err`. Вторая объявляет только `out`, а уже существующей переменной `err` она присваивает значение.

```
in, err := os.Open(infile)
// ...
out, err := os.Create(outfile)
```

Однако краткое объявление переменной должно объявлять по крайней мере одну новую переменную, так что приведенный ниже код не компилируется:

```
f, err := os.Open(infile)
// ...
f, err := os.Create(outfile) // Ошибка: нет новой переменной
```

Чтобы исправить ошибку, во второй инструкции следует использовать обычное присваивание.

Краткое объявление переменной действует как присваивание только для переменных, которые уже были объявлены в том же лексическом блоке; объявления во внешнем блоке игнорируются. Соответствующие примеры приведены в конце этой главы.

2.3.2. Указатели

Переменная представляет собой небольшой блок памяти, содержащий значение. Переменные, созданные с помощью объявлений, идентифицируются по имени, например `x`, но многие переменные идентифицируются только с помощью выражений, таких как `x[i]` или `x.f`. Все эти выражения считывают значение переменной, за исключением ситуаций, когда они находятся слева от оператора присваивания; в этом случае переменной присваивается новое значение.

Значение *указателя* представляет собой *адрес* переменной. Таким образом, указатель — это местоположение в памяти, где хранится значение. Не всякое значение имеет адрес, но его имеет любая переменная. С помощью указателя можно считывать или изменять значение переменной *косвенно*, не используя (и даже не зная) ее имя, если оно у нее есть.

Если переменная объявлена как `var x int`, выражение `&x` ("адрес `x`") дает указатель на целочисленную переменную, т.е. значение типа `*int`, который произносится как "указатель на `int`". Если это значение называется `p`, мы говорим "`p` указывает на

x" или, что то же самое, что "p содержит адрес x". Переменная, на которую указывает p, записывается как *p. Выражение *p дает значение этой переменной, int, но поскольку выражение *p обозначает переменную, оно может использоваться и в левой части присваивания, и в этом случае присваивание обновляет данную переменную.

```
x := 1
p := &x         // p имеет тип *int и указывает на x
fmt.Println(*p) // "1"
*p = 2          // Эквивалентно присваиванию x = 2
fmt.Println(x)  // "2"
```

Каждый компонент переменной составного типа — поле структуры или элемент массива — также является переменной, а значит, имеет свой адрес.

Переменные иногда описываются как *адресуемые* значения. Выражения, которые описывают переменные, являются единственными выражениями, к которым может быть применен оператор *получения адреса* &.

Нулевое значение указателя любого типа равно nil. Проверка p != nil истинна, если p указывает на переменную. Указатели можно сравнивать; два указателя равны тогда и только тогда, когда они указывают на одну и ту же переменную или когда они оба равны nil.

```
var x, y int
fmt.Println(&x == &x, &x == &y, &x == nil) // "true false false"
```

Функция совершенно безопасно может вернуть адрес локальной переменной. Например, в приведенном ниже коде локальная переменная v, созданная этим конкретным вызовом f, будет существовать даже после возврата из функции, и указатель p будет по-прежнему указывать на нее:

```
var p = f()
func f() *int {
    v := 1
    return &v
}
```

Все вызовы f возвращают различные значения:

```
fmt.Println(f() == f()) // "false"
```

Поскольку указатель содержит адрес переменной, передача указателя в функцию в качестве аргумента делает возможным обновление переменной, косвенно переданной в функцию. Например, приведенная ниже функция увеличивает переменную, на которую указывает ее аргумент, и возвращает новое значение переменной, так что оно может использоваться в выражении:

```
func incr(p *int) int {
    *p++  // Увеличивает значение, на которое указывает p;
          // не изменяет значение p
    return *p
}
```

```
v := 1
incr(&v)              // Побочное действие: v теперь равно 2
fmt.Println(incr(&v)) // "3" (и v становится равным 3)
```

Всякий раз, получая адрес переменной или копируя указатель, мы создаем новые *псевдонимы*, или способы идентификации одной и той же переменной. Например, `*p` — это псевдоним для `v`. Псевдонимы-указатели полезны, потому что позволяют нам получить доступ к переменной без использования ее имени, но это палка о двух концах: чтобы найти все инструкции, которые обращаются к переменной, необходимо знать все ее псевдонимы. Это не просто указатели, которые создают псевдонимы; псевдонимы также создаются, когда мы копируем значения других ссылочных типов (наподобие срезов, отображений и каналов), а также структуры, массивы и интерфейсы, которые содержат эти типы.

Указатели являются ключом к пакету `flag`, который использует аргументы командной строки программы для установки значений некоторых переменных, распределенных по всей программе. Для иллюстрации приведенная ниже версия ранее рассмотренной программы `echo` принимает два необязательных флага: `-n` заставляет `echo` пропустить завершающий символ новой строки, который обычно выводится, а `-s sep` заставляет программу разделять выводимые аргументы содержимым строки `sep` вместо используемого по умолчанию одного символа пробела.

gopl.io/ch2/echo4

```
// Echo4 выводит аргументы командной строки.
package main

import (
    "flag"
    "fmt"
    "strings"
)

var n = flag.Bool("n", false, "пропуск символа новой строки")
var sep = flag.String("s", " ", "разделитель")

func main() {
    flag.Parse()
    fmt.Print(strings.Join(flag.Args(), *sep))
    if !*n {
        fmt.Println()
    }
}
```

Функция `flag.Bool` создает новую переменную-флаг типа `bool`. Она принимает три аргумента: имя флага (`"n"`), значение переменной по умолчанию (`false`) и сообщение, которое выводится, если пользователь предоставит неверный аргумент, некорректный флаг или флаг `h` или `help`. Аналогично функция `flag.String` получает имя, значение по умолчанию и сообщение и создает переменную типа `string`. Переменные `sep` и `n` являются указателями на переменные-флаги, доступ к которым осуществляется косвенно, как к `*sep` и `*n`.

Запускаемая программа должна вызвать `flag.Parse` до использования флагов, чтобы переменные-флаги получили новые значения, отличные от значений по умолчанию. Аргументы, не являющиеся флагами, доступны через `flag.Args()` как срез строк. Если функция `flag.Parse` сталкивается с ошибкой, она выводит сообщение об использовании программы и вызывает `os.Exit(2)` для ее завершения.

Давайте немного протестируем программу `echo`.

```
$ go build gopl.io/ch2/echo4
$ ./echo4 a bc def
a bc def
$ ./echo4 s
/ a bc def
a/bc/def
$ ./echo4 n
a bc def
a bc def$
$ ./echo4 help
Использование ./echo4:
   -n    пропуск символа новой строки
   -s string
         разделитель (" " по умолчанию)
```

2.3.3. Функция new

Еще одним способом создания переменной является применение встроенной функции `new`. Выражение `new(T)` создает *неименованную переменную* типа `T`, инициализирует ее нулевым значением типа `T` и возвращает ее адрес, который представляет собой значение типа `*T`.

```
p := new(int)   // p, имеющий тип *int, указывает на неименованную
                // переменную типа int
fmt.Println(*p) // "0"
*p = 2          // Устанавливает значение этой переменной равным 2
fmt.Println(*p) // "2"
```

Переменная, созданная с помощью `new`, ничем не отличается от обычной локальной переменной, у которой берется адрес, за исключением того, что нет необходимости придумывать (и объявлять) ее имя и можно использовать в выражении `new(T)`. Таким образом, `new` является только лишь синтаксическим удобством, но не фундаментальным понятием: две приведенные ниже функции `newInt` имеют идентичное поведение.

```
func newInt() *int {
    return new(int)
}
```

```
func newInt() *int {
    var dummy int
    return &dummy
}
```

Каждый вызов new возвращает отличную от предыдущих переменную с уникальным адресом:

```
p := new(int)
q := new(int)
fmt.Println(p == q) // "false"
```

Существует одно исключение из этого правила: две переменные, тип которых не несет никакой информации, а потому имеющие нулевой размер, такие как struct{} или [0]int, могут, в зависимости от реализации, иметь один и тот же адрес.

Функция new используется относительно редко, поскольку наиболее распространены неименованные переменные структурных типов, для которых имеется более гибкий литеральный синтаксис (раздел 4.4.1).

Поскольку new является предопределенной функцией, а не ключевым словом, это имя можно переопределить для чего-то иного в функции, например:

```
func delta(old, new int) int { return new - old }
```

Конечно, внутри delta встроенная функция new недоступна.

2.3.4. Время жизни переменных

Время жизни переменной — это интервал времени выполнения программы, в течение которого она существует. Время жизни переменной уровня пакета равно времени работы всей программы. Локальные же переменные, напротив, имеют динамическое время жизни: новый экземпляр создается всякий раз, когда выполняется оператор объявления, и переменная живет до тех пор, пока она становится *недоступной*, после чего выделенная для нее память может быть использована повторно. Параметры и результаты функций являются локальными переменными — они создаются всякий раз, когда вызывается их функция.

Например, в фрагменте программы lissajous из раздела 1.4

```
for t := 0.0; t < cycles*2*math.Pi; t += res {
    x := math.Sin(t)
    y := math.Sin(t*freq + phase)
    img.SetColorIndex(size+int(x*size+0.5), size+int(y*size+0.5),
                      blackIndex)
}
```

переменная t создается каждый раз в начале цикла for, а новые переменные x и y создаются на каждой итерации цикла.

Как сборщик мусора узнает, что память, выделенная для хранения переменной, может быть освобождена? Полная история гораздо более длинная и подробная, но основная идея в том, что каждая переменная уровня пакета и каждая локальная переменная каждой в настоящее время активной функции потенциально может быть началом или корнем пути к интересующей нас переменной, который, следуя указателям и другим разновидностям ссылок, в конечном итоге может привести к этой пере-

менной. Если такого пути не существует, переменная недоступна, а потому не может больше влиять на работу программы.

Поскольку время жизни переменной определяется только ее доступностью, локальная переменная может пережить итерацию охватывающего цикла. Она может продолжать существовать даже после возврата из охватывающей функции.

Компилятор может выбрать для выделения памяти для локальных переменных кучу или стек, но каким бы удивительным это ни казалось, этот выбор определяется не тем, что используется для объявления переменной — `var` или `new`.

```
var global *int
func f() {
    var x int
    x = 1
    global = &x
}

func g() {
    y := new(int)
    *y = 1
}
```

Здесь память для переменной `x` должна быть выделена в куче, потому что она остается доступной с помощью переменной `global` после возвращения из `f`, несмотря на ее объявление как локальной переменной. Мы говорим, что `x` *сбегает от* (escapes) `f`. И наоборот, когда выполняется возврат из функции `g`, переменная `*y` становится недоступной, и выделенная для нее память может быть использована повторно. Поскольку `*y` не сбегает от `g`, компилятор может безопасно выделить память для `*y` в стеке, несмотря на то что память выделяется с помощью функции `new`. В любом случае понятие побега не является тем, о чем нужно беспокоиться, чтобы написать правильный код. Тем не менее его хорошо иметь в виду во время оптимизации производительности, поскольку каждая сбегающая переменная требует дополнительной памяти.

Сборка мусора представляет огромную помощь при написании корректных программ, но не освобождает вас от бремени размышлений о памяти. Не нужно явно выделять и освобождать память, но, чтобы писать эффективные программы, все же необходимо знать о времени жизни переменных. Например, сохранение ненужных указателей на короткоживущие объекты внутри долгоживущих объектов, в особенности в глобальных переменных, мешает сборщику мусора освобождать память, выделенную для короткоживущих объектов.

2.4. Присваивания

Значение переменной обновляется с помощью оператора присваивания, который в своей простейшей форме содержит слева от знака = переменную, а справа — выражение.

```
x = 1                       // Именованная переменная
*p = true                   // Косвенная переменная
person.name = "bob"         // Поле структуры
count[x] = count[x]*scale // Элемент массива, среза или отображения
```

Каждый из арифметических или побитовых бинарных операторов имеет соответствующий *присваивающий оператор*, позволяющий, например, переписать последнюю инструкцию как

```
count[x] *= scale
```

Это позволяет не писать лишний раз выражение для переменной.

Числовые переменные могут также быть увеличены или уменьшены с помощью инструкций ++ и --:

```
v := 1
v++ // То же, что и v = v + 1; v становится равным 2
v-- // То же, что и v = v - 1; v становится равным 1
```

2.4.1. Присваивание кортежу

Другая форма присваивания, известная как *присваивание кортежу*, позволяет выполнять присваивание значений нескольким переменным одновременно. Прежде чем любая из переменных в левой части получит новое значение, вычисляются все выражения из правой части, что делает эту разновидность присваивания наиболее полезной, когда некоторые переменные встречаются с обеих сторон оператора присваивания, как это происходит, например, при обмене значений двух переменных:

```
x, y = y, x
a[i], a[j] = a[j], a[i]
```

Или при вычислении наибольшего общего делителя двух целых чисел:

```
func gcd(x, y int) int {
    for y != 0 {
        x, y = y, x%y
    }
    return x
}
```

Или при итеративном вычислении n – 1-го числа Фибоначчи:

```
func fib(n int) int {
    x, y := 0, 1
    for i := 0; i < n; i++ {
        x, y = y, x+y
    }
    return x
}
```

Присваивание кортежу может также сделать последовательность тривиальных присваиваний более компактной:

```
i, j, k = 2, 3, 5
```

хотя (в качестве вопроса стиля) следует избегать присваивания кортежам при наличии сложных выражений: последовательность отдельных инструкций читается легче.

Некоторые выражения, такие как вызов функции с несколькими результатами, производят несколько значений. Когда такой вызов используется в инструкции присваивания, в левой части должно быть столько переменных, сколько результатов возвращает функция:

```
f, err = os.Open("foo.txt") // Вызов функции возвращает два значения
```

Зачастую функции используют дополнительные результаты для указания происшедшей ошибки, возвращая значение `error`, как в случае вызова `os.Open`, либо значение типа `bool`, обычно с именем `ok`. Как мы увидим в последующих главах, есть три оператора, которые иногда ведут себя таким образом. Если в присваивании, в котором ожидаются два результата, находится поиск в отображении (раздел 4.3), утверждение о типе (раздел 7.10) или получение из канала (раздел 8.4.2), создается дополнительный логический результат:

```
v, ok = m[key] // Поиск в отображении
v, ok = x.(T)  // Утверждение о типе
v, ok = <-ch   // Получение из канала
```

Как и в случае объявления переменных, мы можем присвоить ненужные значения пустому идентификатору:

```
_, err = io.Copy(dst, src) // Отбрасываем количество байтов
_, ok = x.(T)              // Проверка типа, игнорируем результат
```

2.4.2. Присваиваемость

Операторы присваивания представляют собой явную форму присваивания, но в программе есть много мест, в которых присваивание происходит *неявно*: вызов функции неявно присваивает значения аргументов соответствующим переменным параметров; инструкция `return` неявно присваивает операнды `return` соответствующим результирующим переменным; литеральное выражение для составного типа (раздел 4.2), такое, как срез

```
medals := []string{"gold", "silver", "bronze"}
```

неявно присваивает каждый элемент, как если бы оно было записано как

```
medals[0] = "gold"
medals[1] = "silver"
medals[2] = "bronze"
```

Элементы отображений и каналы, хотя и не являются обычными переменными, также оказываются субъектами подобных неявных присваиваний.

Присваивание, явное или неявное, всегда корректно, если левая часть (переменная) и правая часть (значение) имеют один и тот же тип. Говоря более общо, присваивание разрешено, только если значение *присваиваемо* (assignable) типу переменной.

Правило *присваиваемости* имеет свои частные случаи для разных типов, так что мы будем их рассматривать по мере рассмотрения новых типов. Для типов, которые рассматривались до сих пор, правило очень простое: типы должны точно соответствовать, но значение `nil` может быть присвоено любой переменной типа интерфейса или ссылочного типа. Константы (раздел 3.6) имеют более гибкие правила присваиваемости, которые позволяют избежать большинства явных преобразований.

Вопрос о том, можно ли сравнивать два значения с помощью операторов `==` и `!=`, связан с присваиваемостью: в любых сравнениях первый операнд должен быть присваиваемым типу второго операнда, и наоборот. Как и в случае присваиваемости, мы будем рассматривать соответствующие случаи правил *сравниваемости* при представлении каждого нового типа.

2.5. Объявления типов

Тип переменной или выражения определяет характеристики значений, которые он может принимать, такие как их размер (количество битов или, возможно, количество элементов), как они представлены внутренне, операции, которые могут быть над ними выполнены, и связанные с ними методы.

В любой программе есть переменные, которые используют одинаковое внутреннее представление, но значительно различаются в смысле представляемой ими концепции. Например, `int` может использоваться для представления индекса цикла, временной метки, дескриптора файла или месяца; `float64` может представлять скорость в метрах в секунду или температуру в одной из нескольких шкал; `string` может быть паролем или названием цвета.

Объявление `type` определяет новый *именованный тип*, который имеет тот же *базовый тип*, что и существующий. Именованный тип обеспечивает возможность отличать различные и, возможно, несовместимые использования базового типа с тем, чтобы они не могли оказаться непреднамеренно смешанными.

```
type имя базовый_тип
```

Объявления типов наиболее часто появляются на уровне пакета, где именованный тип виден всему пакету, и если это имя экспортируется (начинается с прописной буквы), то оно доступно и в других пакетах.

Для иллюстрации объявлений типов давайте превратим различные температурные шкалы в разные типы.

gopl.io/ch2/tempconv0

```
// Пакет tempconv выполняет вычисления температур
// по Цельсию (Celsius) и по Фаренгейту (Fahrenheit).
package tempconv

import "fmt"

type Celsius    float64
type Fahrenheit float64
```

```
const (
    AbsoluteZeroC Celsius = -273.15
    FreezingC     Celsius = 0
    BoilingC      Celsius = 100
)

func CToF(c Celsius) Fahrenheit { return Fahrenheit(c*9/5 + 32) }
func FToC(f Fahrenheit) Celsius { return Celsius((f - 32)*5/9) }
```

Этот пакет определяет два типа, `Celsius` и `Fahrenheit`, для двух шкал температуры. Несмотря на то что оба они имеют один и тот же базовый тип, `float64`, сами они являются разными типами, а потому не могут сравниваться или объединяться в арифметических выражениях. Такое различение типов позволяет избежать ошибок непреднамеренного сочетания температур в двух различных масштабах; требуется *явное* преобразование типа для значения `float64`, такое как `Celsius(t)` или `Fahrenheit(t)`. Это именно преобразования типа, а не вызовы функций `Celsius(t)` и `Fahrenheit(t)`. Они никоим образом не изменяют значение или его представление, но явно изменяют его смысл. С другой стороны, функции `CToF` и `FToC` выполняют преобразование температур между двумя различными шкалами; они возвращают *отличающиеся* от исходных значения.

Для каждого типа `T` имеется соответствующая операция преобразования `T(x)`, которая приводит значение `x` к типу `T`. Преобразование одного типа в другой разрешено, если оба они имеют один и тот же базовый тип или если оба они являются неименованными указателями на переменные одного и того же базового типа; такие преобразования изменяют тип, но не представление значения. Если `x` присваиваемо типу `T`, то преобразование разрешено, но обычно является излишним.

Как мы увидим в следующей главе, разрешены также преобразования между числовыми типами и между строками и некоторыми типами срезов. Эти преобразования могут изменить представление значения. Например, преобразование числа с плавающей запятой в целое отбрасывает дробную часть, а преобразование строки в срез `[]byte` выделяет память для копии данных строки. В любом случае преобразование никогда не приводит к сбою времени выполнения программы.

Базовый тип для именованного типа определяет его структуру и представление, а также набор поддерживаемых внутренних операций, которые являются такими же, как и при непосредственном использовании базового типа. Это означает, что арифметические операторы одинаково работают для `Celsius` и `Fahrenheit` и точно так же, как и для `float64`, чего и следовало ожидать.

```
fmt.Printf("%g\n", BoilingCFreezingC)         // "100" °C
boilingF := CToF(BoilingC)
fmt.Printf("%g\n", boilingFCToF( FreezingC))  // "180" °F
fmt.Printf("%g\n", boilingFFreezingC)         // Ошибка компиляции:
                                              // несоответствие типов
```

Операторы сравнения, такие как == и <, могут использоваться для сравнения значения именованного типа с другим значением того же типа или со значением неиме-

нованного типа с тем же базовым типом. Но два значения различных именованных типов сравнивать непосредственно нельзя:

```
var c Celsius
var f Fahrenheit
fmt.Println(c == 0)   // "true"
fmt.Println(f >= 0)   // "true"
fmt.Println(c == f)   // Ошибка компиляции: несоответствие типов
fmt.Println(c == Celsius(f)) // "true"!
```

Внимательно рассмотрите последний случай. Несмотря на свое название, преобразование типа `Celsius(f)` не изменяет значение своего аргумента, а изменяет только его тип. Проверка дает значение `true`, потому что и `c`, и `f` равны нулю.

Именованный тип предоставляет определенные удобства, помогая избежать написания сложных типов снова и снова. Это преимущество почти не проявляется, когда базовый тип такой простой, как `float64`, но оказывается существенным для сложных типов, в чем мы убедимся, когда будем рассматривать структуры.

Именованные типы позволяют также определить новое поведение значений этого типа. Такое поведение выражается в виде набора функций, связанных с данным типом и именуемых *методами* типа. Методы подробно описываются в главе 6, "Методы", но уже сейчас мы попробуем этот механизм "на вкус".

Приведенное далее объявление, в котором параметра `c` типа `Celsius` находится перед именем функции, ассоциирует с типом `Celsius` метод с именем `String`, который возвращает числовое значение `c`, за которым следуют символы `°C`:

```
func (c Celsius) String() string { return fmt.Sprintf("%g°C", c) }
```

Многие типы объявляют метод `String` такого вида, поскольку, как мы увидим в разделе 7.1, он управляет видом значений данного типа при выводе строк пакетом `fmt`.

```
c := FToC(212.0)
fmt.Println(c.String()) // "100°C"
fmt.Printf("%v\n", c)   // "100°C"; явный вызов String не нужен
fmt.Printf("%s\n", c)   // "100°C"
fmt.Println(c)          // "100°C"
fmt.Printf("%g\n", c)   // "100"; не вызывает String
fmt.Println(float64(c)) // "100"; не вызывает String
```

2.6. Пакеты и файлы

Пакеты в Go служат тем же целям, что и библиотеки или модули в других языках программирования, поддерживая модульность, инкапсуляцию, раздельную компиляцию и повторное использование. Исходный текст пакета располагается в одном или нескольких файлах `.go`, обычно в каталоге, имя которого является окончанием пути импорта, например файлы пакета `gopl.io/ch1/helloworld` располагаются в каталоге `$GOPATH/src/gopl.io/ch1/helloworld`.

Каждый пакет служит в качестве отдельного *пространства имен* для своих объявлений. В пакете `image`, например, идентификатор `Decode` относится к функции, отличной от той, на которую указывает идентификатор в пакете `unicode/utf16`. Чтобы обратиться к функции за пределами ее пакета, следует *квалифицировать* идентификатор, явно указав, что имеется в виду: `image.Decode` или `utf16.Decode`.

Пакеты также позволяют скрывать информацию, управляя тем, какие имена видны за пределами пакета (*экспортированы*). В Go тем, какие идентификаторы экспортируются, а какие — нет, управляет очень простое правило: экспортируются идентификаторы, которые начинаются с прописной буквы.

Чтобы проиллюстрировать эти основы, предположим, что наше программное обеспечение для преобразования температур стало популярным и что мы хотим сделать его доступным для сообщества Go в виде нового пакета. Как мы это сделаем?

Давайте создадим пакет `gopl.io/ch2/tempconv`, представляющий собой вариацию предыдущего примера. (Здесь мы сделали исключение для нашего обычного правила последовательной нумерации примеров, чтобы путь пакета был более реалистичным.) Сам пакет хранится в двух файлах, чтобы показать доступность объявлений в отдельных файлах пакета; в реальности такой крошечный пакет, как рассматриваемый нами, вполне поместится в одном файле.

Поместим объявления типов, констант и методов в файл `tempconv.go`:

gopl.io/ch2/tempconv

```
// Пакет tempconv выполняет преобразования температур.
package tempconv

import "fmt"

type Celsius float64
type Fahrenheit float64

const (
    AbsoluteZeroC Celsius = -273.15
    FreezingC     Celsius = 0
    BoilingC      Celsius = 100
)

func (c Celsius) String() string    { return fmt.Sprintf("%g°C", c) }
func (f Fahrenheit) String() string { return fmt.Sprintf("%g°F", f) }
```

а функции преобразования — в файл `conv.go`:

```
package tempconv
// CToF преобразует температуру по Цельсию в температуру по Фаренгейту.
func CToF(c Celsius) Fahrenheit { return Fahrenheit(c*9/5 + 32) }
// FToC преобразует температуру по Фаренгейту в температуру по Цельсию.
func FToC(f Fahrenheit) Celsius { return Celsius((f-32)*5/9) }
```

Каждый файл начинается с объявления `package`, которое определяет имя пакета. При импорте пакета к его членам следует обращаться как к `tempconv.CToF`, и т.д. Имена уровня пакета, такие как типы и константы, объявленные в одном файле паке-

та, являются видимыми для всех других файлов пакета, как если бы весь исходный текст располагался в единственном файле. Обратите внимание, что `tempconv.go` импортирует `fmt`, но `conv.go` этого не делает, так как не использует ничего из пакета `fmt`.

Поскольку имена констант уровня пакета начинаются с прописных букв, они доступны и с помощью квалифицированных имен, таких как `tempconv.AbsoluteZeroC`:

```
fmt.Printf("Брррр! %v\n", tempconv.AbsoluteZeroC) // "Брррр! -273.15° C"
```

Чтобы преобразовать температуру по Цельсию в температуру по Фаренгейту в пакете, который импортирует `gopl.io/ch2/tempconv`, можно написать следующий код:

```
fmt.Println(tempconv.CToF(tempconv.BoilingC)) // "212°F"
```

Комментарий, непосредственно предшествующий объявлению `package`, документирует пакет в целом. По соглашению он должен начаться с резюме в проиллюстрированном в примере стиле. В каждом пакете только один файл должен иметь такой *документирующий комментарий* (раздел 10.7.4). Обширные документирующие комментарии часто помещаются в отдельный файл, по соглашению именуемый `doc.go`.

Упражнение 2.1. Добавьте в пакет `tempconv` типы, константы и функции для работы с температурой по шкале Кельвина, в которой нуль градусов соответствует температуре –273.15°C, а разница температур в 1K имеет ту же величину, что и 1°C.

2.6.1. Импорт

В программе Go каждый пакет идентифицируется уникальной строкой, которая называется его *путем импорта*. Это строки, которые появляются в объявлении `import`, такие как `"gopl.io/ch2/tempconv"`. Спецификация языка не определяет, откуда берутся эти строки или что они означают; их интерпретация — дело соответствующего инструментария. При использовании инструмента `go` (глава 10, "Пакеты и инструменты Go") путь импорта обозначает каталог, содержащий один или несколько исходных файлов Go, совместно образующих пакет.

В дополнение к пути импорта каждый пакет имеет *имя пакета*, которое является кратким (и не обязательно уникальным) именем и находится в объявлении `package`. По соглашению имя пакета соответствует последней части пути импорта, так что легко предсказать, что именем пакета `gopl.io/ch2/tempconv` является `tempconv`.

Чтобы воспользоваться пакетом `gopl.io/ch2/tempconv`, его следует импортировать.

gopl.io/ch2/cf

```
// Cf конвертирует числовой аргумен в температуру
// по Цельсию и по Фаренгейту.
package main

import (
    "fmt"
    "gopl.io/ch2/tempconv"
```

```
    "os"
    "strconv"
)

func main() {
    for _, arg := range os.Args[1:] {
        t, err := strconv.ParseFloat(arg, 64)
        if err != nil {
            fmt.Fprintf(os.Stderr, "cf: %v\n", err)
            os.Exit(1)
        }
        f := tempconv.Fahrenheit(t)
        c := tempconv.Celsius(t)
        fmt.Printf("%s = %s, %s = %s\n",
            f, tempconv.FToC(f), c, tempconv.CToF(c))
    }
}
```

Объявление импорта связывает краткое имя импортируемого пакета, которое может использоваться для обращения к его содержимому в данном файле. Показанное выше объявление `import` позволяет обращаться к именам в `gopl.io/ch2/tempconv` с помощью *квалифицированных идентификаторов* наподобие `tempconv.CToF`. По умолчанию краткое имя представляет собой имя пакета (в данном случае — `tempconv`), но объявление импорта может указать и альтернативное имя, чтобы избежать конфликтов (раздел 10.4).

Программа `cf` преобразует один числовой аргумент командной строки в значения по Цельсию и по Фаренгейту:

```
$ go build gopl.io/ch2/cf
$ ./cf 32
32°F = 0°C, 32°C = 89.6°F
$ ./cf 212
212°F = 100°C, 212°C = 413.6°F
$ ./cf -40
-40°F = -40°C, -40°C = -40°F
```

Импорт пакета без последующего его использования считается ошибкой. Такая проверка помогает избежать зависимостей, которые становятся ненужными по мере развития кода (хотя это может быть неудобным во время отладки, поскольку, например, закомментированная строка кода наподобие `log.Print("got here!")` может убрать единственную ссылку на пакет `log` и привести тем самым к выводу компилятором сообщения об ошибке. В этой ситуации необходимо одновременно закомментировать (или удалить) ненужные объявления `import`.

Еще лучше — воспользоваться инструментом `golang.org/x/tools/cmd/goimports`, который при необходимости автоматически добавляет и удаляет пакеты из объявлений импорта; большинство редакторов может быть настроено для запуска

goimports при каждом сохранении файла. Подобно инструменту gofmt, goimports также приводит исходные тексты Go к каноническому формату.

Упражнение 2.2. Напишите программу общего назначения для преобразования единиц, аналогичную cf, которая считывает числа из аргументов командной строки (или из стандартного ввода, если аргументы командной строки отсутствуют) и преобразует каждое число в другие единицы, как температуру — в градусы Цельсия и Фаренгейта, длину — в футы и метры, вес — в фунты и килограммы и т.д.

2.6.2. Инициализация пакетов

Инициализации пакета начинается с инициализации переменных уровня пакета — в том порядке, в котором они объявлены, с тем исключением, что первыми разрешаются зависимости:

```
var a = b + c // a инициализируется третьей, значением 3
var b = f()   // b инициализируется второй, значением 2 из вызова f
var c = 1     // c инициализируется первой, значением 1

func f() int { return c + 1 }
```

Если пакет состоит из нескольких файлов .go, они инициализируются в том порядке, в каком файлы передаются компилятору; инструмент go перед вызовом компилятора сортирует файлы .go по имени.

Каждая переменная, объявленная на уровне пакета, начинает существование со значением, равным значению инициализирующего выражения (если таковое имеется); но для некоторых переменных, таких как таблицы данных, инициализирующее выражение может быть не самым простым средством задания начального значения. В этом случае проще может оказаться механизм функции init. Любой файл может содержать любое количество функций, объявление которых имеет вид

```
func init() { /* ... */ }
```

Такие функции init нельзя вызвать или обратиться к ним, но во всех прочих отношениях они являются нормальными функциями. В каждом файле функции init выполняются автоматически при запуске программы в том порядке, в котором они объявлены.

Инициализация выполняется по одному пакету за раз в порядке объявлений импорта в программе; первыми обрабатываются зависимости, поэтому пакет p, импортирующий пакет q, может быть уверен, что q будет полностью инициализирован до начала инициализации p. Инициализация выполняется снизу вверх; последним инициализируется пакет main. Таким образом, все пакеты оказываются полностью инициализированными до начала выполнения функции main приложения.

Приведенный ниже пакет определяет функцию PopCount, которая возвращает число установленных битов (т.е. битов, значение которых равно 1) в значение типа uint64. Она использует функцию init для предварительного вычисления таблицы результатов pc для всех возможных 8-битовых значений, так что функции PopCount

нужно не выполнять 64 шага, а просто вернуть сумму восьми значений из таблицы. (Это, безусловно, не самый быстрый алгоритм подсчета битов, но он очень удобен для иллюстрации функций `init` и демонстрации такого полезного приема программирования, как предварительное вычисление таблицы значений.)

gopl.io/ch2/popcount

```
package popcount

// pc[i] — количество единичных битов в i.
var pc [256]byte

func init() {
    for i := range pc {
        pc[i] = pc[i/2] + byte(i&1)
    }
}

// PopCount возвращает степень заполнения
// (количество установленных битов) значения x.
func PopCount(x uint64) int {
    return int(pc[byte(x>>(0*8))] +
        pc[byte(x>>(1*8))] +
        pc[byte(x>>(2*8))] +
        pc[byte(x>>(3*8))] +
        pc[byte(x>>(4*8))] +
        pc[byte(x>>(5*8))] +
        pc[byte(x>>(6*8))] +
        pc[byte(x>>(7*8))])
}
```

Обратите внимание, что цикл `range` в функции `init` использует только индекс; значение не является необходимым и потому не включено. Этот цикл можно также записать следующим образом:

```
for i, _ := range pc {
```

Другие применения функций `init` мы рассмотрим в следующем разделе и в разделе 10.5.

Упражнение 2.3. Перепишите функцию `PopCount` так, чтобы она использовала цикл вместо единого выражения. Сравните производительность двух версий. (В разделе 11.4 показано, как правильно сравнивать производительность различных реализаций.)

Упражнение 2.4. Напишите версию `PopCount`, которая подсчитывает биты с помощью сдвига аргумента по всем 64 позициям, проверяя при каждом сдвиге крайний справа бит. Сравните производительность этой версии с выборкой из таблицы.

Упражнение 2.5. Выражение `x&(x-1)` сбрасывает крайний справа ненулевой бит `x`. Напишите версию `PopCount`, которая подсчитывает биты с использованием этого факта, и оцените ее производительность.

2.7. Область видимости

Объявление связывает имя с сущностью в программе, такой как функция или переменная. *Область видимости* является частью исходного кода, в которой использование объявленного имени ссылается на сущность из этого объявления.

Не путайте область видимости и время жизни. Область видимости — это область исходного текста программы; это свойство времени компиляции. Временем же жизни переменной называют диапазон времени выполнения, когда к переменной можно обращаться из других частей программы; это свойство времени выполнения.

Синтаксический *блок* представляет собой последовательность инструкций, заключенных в фигурные скобки, подобные тем, которые окружают тело функции или цикла. Имя, объявленное внутри синтаксического блока, не видимо вне блока. Блок охватывает свои объявления и определяет их область видимости. Мы можем обобщить это понятие блоков, включив в него другие группы объявлений, которые не охватываются фигурными скобками в исходном тексте явно; будем называть их *лексическими блоками*. Имеются лексический блок для всего исходного текста, именуемый *всеобщим блоком* (universe block); блок для каждого пакета; для каждого файла; для каждой инструкции `for`, `if` и `switch`; для каждого `case` в конструкции `switch` или `select`; и, конечно, для каждого явного синтаксического блока.

Лексический блок объявления определяет его область видимости, которая может быть большой или малой. Объявления встроенных типов, функций и констант наподобие `int`, `len` и `true` находятся во всеобщем блоке, и обратиться к ним можно на протяжении всей программы. К объявлениям вне любой функции, т.е. на *уровне пакета*, можно обратиться из любого файла в том же самом пакете. Импортированные пакеты, такие как `fmt` в примере `tempconv`, объявляются на *уровне файлов*, так что к ним можно обращаться из того же файла, но не из другого файла в том же пакете без отдельной директивы `import`. Многие объявления, подобно объявлению переменной `c` в функции `tempconv.CToF`, являются *локальными*, а потому обращаться к ним можно только в пределах той же функции (или, возможно, только ее части).

Областью видимости метки управления потоком, используемой инструкциями `break`, `continue` и `goto`, является вся охватывающая функция.

Программа может содержать несколько объявлений одного и того же имени при условии, что все объявления находятся в различных лексических блоках. Например, можно объявить локальную переменную с тем же именем, что и переменная уровня пакета. Или, как показано в разделе 2.3.3, можно объявить параметр функции с именем `new`, даже несмотря на то, что функция с этим именем является объявленной во всеобщем блоке. Но не переусердствуйте; чем больше область видимости переопределения, тем больше шансов запутать читателя исходного текста.

Компилятор, встретив ссылку на имя, ищет объявление, начиная с наиболее глубоко вложенного внутреннего лексического блока и продолжая до всеобщего блока. Если компилятор не обнаруживает объявление, он сообщает об ошибке "необъявленное имя". Если имя объявлено и в наружном, и во внутреннем блоках, первым будет обнаружено внутреннее объявление. В этом случае говорят, что внутреннее объявление *затеняет* (shadow) или *скрывает* (hide) внешнее, делая его недоступным:

```
func f() {}

var g = "g"

func main() {
    f := "f"
    fmt.Println(f) // "f"; локальная переменная f затеняет
                   // функцию f уровня пакета
    fmt.Println(g) // "g"; переменная уровня пакета
    fmt.Println(h) // Ошибка компиляции: неопределенное имя h
}
```

Внутри функции лексические блоки могут быть вложенными с произвольной глубиной вложения, поэтому одно локальное объявление может затенять другое. Большинство блоков создаются конструкциями управления потоком, такими как инструкции `if` и циклы `for`. Приведенная ниже программа имеет три различных переменных `x`, так как все объявления появляются в различных лексических блоках. (Этот пример иллюстрирует правила областей видимости, но не хороший стиль программирования!)

```
func main() {
    x := "hello!"
    for i := 0; i < len(x); i++ {
        x := x[i]
        if x != '!' {
            x := x + 'A' - 'a'
            fmt.Printf("%c", x) // "HELLO" (по букве за итерацию)
        }
    }
}
```

Выражения `x[i]` и `x + 'A' - 'a'` ссылаются на объявления `x` из внешнего блока; мы поясним их чуть позже. (Обратите внимание, что второе выражение *не* эквивалентно вызову `unicode.ToUpper`.)

Как упоминалось выше, не все лексические блоки соответствуют последовательностям операторов, помещенным в фигурные скобки; некоторые из них просто подразумеваются. Цикл `for` в исходном тексте выше создает два лексических блока: явный блок для тела цикла и неявный блок, который дополнительно охватывает переменные, объявленные в инициализации цикла (в данном случае — переменную `i`). Областью видимости переменной, объявленной в неявном блоке, являются условие, последействие (`i++`) и тело цикла `for`.

В приведенном ниже примере также имеется три переменные с именем `x`, каждая из которых объявлена в своем блоке: одна — в теле функции, другая — в блоке цикла `for` и третья — в теле цикла (но только два из этих блоков указаны явно):

```
func main() {
    x := "hello"
    for _, x := range x {
```

```
        x := x + 'A' - 'a'
        fmt.Printf("%c", x) // "HELLO" (по букве за итерацию)
    }
}
```

Подобно циклам `for`, инструкции `if` и `switch` также создают неявные блоки в дополнение к блокам их тел. Код в следующей цепочке `if-else` показывает области видимости `x` и `y`:

```
if x := f(); x == 0 {
    fmt.Println(x)
} else if y := g(x); x == y {
    fmt.Println(x, y)
} else {
    fmt.Println(x, y)
}
fmt.Println(x, y) // Ошибка компиляции: x и y здесь невидимы
```

Вторая инструкция `if` вложена в первую, так что переменные, объявленные внутри инициализатора первой инструкции, видимы внутри второй инструкции. Аналогичные правила применяются и для каждого `case` инструкции `switch`: имеется блок для условия и блок для каждого тела `case`.

На уровне пакета порядок, в котором появляются объявления, не влияет на их область видимости, поэтому объявление может ссылаться само на себя или на другое объявление, которое следует за ним, позволяя нам объявлять рекурсивные или взаимно рекурсивные типы и функции. Однако компилятор сообщит об ошибке, если объявление константы или переменной ссылается само на себя.

```
if f, err := os.Open(fname); err != nil { // Ошибка компиляции: неис-
    return err                            // пользованная переменная f
}
f.Stat()     // Ошибка компиляции: неопределенная переменная f
f.Close()    // Ошибка компиляции: неопределенная переменная f
```

В приведенной выше программе областью видимости `f` является только инструкция `if`, так что переменная `f` недоступна для инструкций, следующих далее, что приводит к ошибке компиляции. В зависимости от используемого компилятора вы можете получить дополнительное сообщение об ошибке, гласящее о том, что переменная `f` нигде не используется.

Таким образом, часто приходится объявлять переменную `f` до условия, чтобы она была доступна после него:

```
f, err := os.Open(fname)
if err != nil {
    return err
}
f.Stat()
f.Close()
```

Может показаться соблазнительной возможность избежать объявления `f` и `err` во внешнем блоке, перемещая вызовы методов `Stat` и `Close` внутрь блока `else`:

```
if f, err := os.Open(fname); err != nil {
    return err
} else {
    // f и err видимы здесь
    f.Stat()
    f.Close()
}
```

Однако обычная практика программирования в Go заключается в работе с ошибкой в конструкции `if` с последующим выходом из функции так, чтобы успешный путь выполнения не сопровождался отступом.

Краткие объявления переменных требуют понимания области видимости. Рассмотрим программу, приведенную ниже, которая начинается с получения текущего рабочего каталога и его сохранения в переменной уровня пакета. Это может быть сделано с помощью вызова `os.Getwd` в функции `main`, но, может быть, лучше отделить это действие от основной логики программы, в особенности если невозможность получить рабочий каталог является фатальной ошибкой. Функция `log.Fatalf` выводит соответствующее сообщение и вызывает `os.Exit(1)`.

```
var cwd string
func init() {
    cwd, err := os.Getwd() // Ошибка компиляции: cwd не используется
    if err != nil {
        log.Fatalf("Ошибка os.Getwd: %v", err)
    }
}
```

Поскольку ни `cwd`, ни `err` не объявлены в блоке функции `init`, оператор `:=` объявляет обе эти переменные как локальные. Внутреннее объявление `cwd` делает внешнее недоступным, поэтому данная инструкция не обновляет значение переменной уровня пакета `cwd`, как предполагалось.

Нынешние компиляторы Go обнаруживают, что локальная переменная `cwd` никогда не используется, и сообщают об этом как об ошибке, но они не обязаны выполнять эту проверку в обязательном порядке. Кроме того, незначительные изменения, такие как добавление инструкции записи в журнал значения локальной переменной `cwd`, лишили бы эту проверку смысла.

```
var cwd string
func init() {
    cwd, err := os.Getwd() // Примечание: неверно!
    if err != nil {
        log.Fatalf("Ошибка os.Getwd: %v", err)
    }
    log.Printf("Рабочий каталог = %s", cwd)
}
```

Глобальная переменная cwd остается неинициализированной, и кажущийся корректным вывод в журнал только скрывает ошибку.

Существует ряд способов справиться с этой потенциальной проблемой. Самый простой из них — избежать оператора :=, объявив err в отдельном объявлении var:

```
var cwd string
func init() {
    var err error
    cwd, err = os.Getwd()
    if err != nil {
        log.Fatalf("Ошибка os.Getwd failed: %v", err)
    }
}
```

Теперь мы видели, как пакеты, файлы, объявления и инструкции выражают структуру программ. Следующие две главы мы посвятим структурам данных.

3

Фундаментальные типы данных

В конечном счете вся память представляет собой просто биты, но компьютер работает с их фиксированными количествами, именуемыми *словами*, которые интерпретируются как целые числа, числа с плавающей точкой, множества битов и адреса памяти, а затем объединяются в большие агрегаты, которые представляют пакеты, пиксели, портфолио, стихи или что-то еще. Go предлагает ряд способов организации данных с широким спектром типов данных, на одном конце которого находятся типы, соответствующие возможностям аппаратного обеспечения, а на другом — все, что нужно для того, чтобы программистам было удобно представлять сложные структуры данных.

Типы Go делятся на четыре категории: *фундаментальные типы*, *составные* (*агрегированные*) *типы*, *ссылочные типы* и *типы интерфейсов*. Фундаментальные типы, которые являются темой данной главы, включают числа, строки и булевы значения. Составные типы — массивы (раздел 4.1) и структуры (раздел 4.4) — образуют более сложные типы данных путем объединения более простых. Ссылочные типы представляют собой разнородную группу, которая включает указатели (раздел 2.3.2), срезы (раздел 4.2), отображения (раздел 4.3), функции (глава 5, "Функции") и каналы (глава 8, "Go-подпрограммы и каналы"), но общим у них является то, что все они обращаются к переменным или состояниям программы *косвенно*, так что результат действия операции, примененной к одной ссылке, наблюдается всеми ее копиями. О типах интерфейсов мы поговорим в главе 7, "Интерфейсы".

3.1. Целые числа

Числовые типы данных Go включают целые числа нескольких размеров, числа с плавающей точкой и комплексные числа. Каждый числовой тип определяет размер и знаковость своих значений. Начнем с целых чисел.

Go обеспечивает как знаковую, так и беззнаковую целочисленную арифметику. Имеются знаковые целые числа четырех размеров — 8, 16, 32 и 64 бита, — представленные типами `int8`, `int16`, `int32` и `int64`, а также соответствующие беззнаковые версии `uint8`, `uint16`, `uint32` и `uint64`.

Есть также два типа, которые называются просто `int` и `uint` и имеют естественный или наиболее эффективный размер для знаковых и беззнаковых целых чисел на конкретной платформе; `int` на сегодняшний день является наиболее широко используемым числовым типом. Оба эти типа имеют одинаковый размер (либо 32, либо 64 бита), но вы не должны делать никаких предположений о том, какой именно из них используется; различные компиляторы могут делать разный выбор размера даже на одном и том же аппаратном обеспечении.

Тип `rune` является синонимом для типа `int32` и по соглашению указывает, что данное значение является символом Unicode. Эти два имени могут использоваться взаимозаменяемо. Аналогично тип `byte` является синонимом для типа `uint8` и подчеркивает, что это значение является фрагментом неформатированных данных, а не малым числом.

Наконец имеется беззнаковый целочисленный тип `uintptr`, ширина которого не указана, но достаточна для хранения всех битов значения указателя. Тип `uintptr` используется только для низкоуровневого программирования, такого, как стыковка программы Go с библиотекой C или с операционной системой. Мы увидим соответствующие примеры, когда доберемся до пакета `unsafe` в главе 13, "Низкоуровневое программирование".

Независимо от размера типы `int`, `uint` и `uintptr` отличаются от своих "братьев" с явно указанным размером. Таким образом, `int` — это не тот же тип, что и `int32`, даже если на данной платформе естественный размер целых чисел составляет 32 бита и там, где нужно использовать значение типа `int`, в то время как требуется значение типа `int32` (и наоборот), необходимо явное преобразование типов.

Знаковые числа представлены в формате дополнения до 2, в котором старший бит зарезервирован для знака числа, а диапазон значений n-разрядного числа простирается от -2^{n-1} до $2^{n-1}-1$. Целые беззнаковые числа используют для представления неотрицательных значений все биты и, таким образом, имеют диапазон значений от 0 до 2^n-1. Например, диапазон значений `int8` простирается от –128 до 127, тогда как диапазон `uint8` — от 0 до 255.

Ниже перечислены бинарные операторы Go (арифметические, логические и операторы сравнения) в порядке уменьшения приоритета.

```
*     /     %     <<    >>    &     &^
+     -     |     ^
==    !=    <     <=    >     >=
&&
||
```

Есть только пять уровней приоритета для бинарных операторов. Операторы на одном и том же уровне левоассоциативны, поэтому для ясности или для того, чтобы действия выполнялись в нужном порядке, могут быть необходимы скобки, как, например, в выражении наподобие `mask & (1 << 28)`.

Каждый оператор в первых двух строках таблицы выше, например `+`, имеет соответствующий *присваивающий оператор*, такой как `+=`, который может использоваться для сокращения записи инструкции присваивания.

Арифметические операторы +, -, * и / могут применяться к целым числам, числам с плавающей точкой и комплексным числам, но оператор получения остатка при делении % применяется только к целым числам. Поведение % для отрицательных чисел различно в разных языках программирования. В Go знак остатка всегда такой же, как и знак делимого, так что и `-5%3`, и `-5%-3` равны `-2`. Поведение оператора / зависит от того, являются ли его операнды целыми числами, так что `5.0/4.0` равно `1.25`, но `5/4` равно `1`, так как при целочисленном делении происходит усечение результата по направлению к нулю.

Если результат арифметической операции, как знаковой, так и беззнаковой, имеет больше битов, чем может быть представлено типом результата, мы говорим о *переполнении* (overflow). При этом старшие биты, которые не помещаются в результате, молча отбрасываются. Если исходное число имеет знаковый тип, результат может быть отрицательным, если левый бит равен `1`, как показано в следующем примере с `int8`:

```
var u uint8 = 255
fmt.Println(u, u+1, u*u) // "255 0 1"
var i int8 = 127
fmt.Println(i, i+1, i*i) // "127 -128 1"
```

Два целых числа одного и того же типа можно сравнить с использованием бинарных операторов сравнения, показанных ниже; результат выражения сравнения является логическим (булевым) значением.

`==`	Равно
`!=`	Не равно
`<`	Меньше, чем
`<=`	Меньше или равно
`>`	Больше, чем
`>=`	Больше или равно

Фактически все значения фундаментальных типов — логические, числа и строки — являются *сравниваемыми*, а это означает, что два значения одного и того же типа можно сравнивать с помощью операторов == и !=. Кроме того, целые числа, числа с плавающей точкой и строки являются *упорядочиваемыми* с помощью операторов сравнения. Значения многих других типов не являются сравниваемыми, а многие другие типы не являются упорядочиваемыми. По ходу знакомства с разными типами мы будем рассматривать правила, регулирующие *сравниваемость* их значений.

Имеются также унарные операторы + и -.

`+`	Унарный знак плюс (не оказывает никакого действия)
`-`	Унарный знак минус

Для целых чисел запись `+x` является краткой формой записи для `0+x`, а `-x` — краткой формой для `0-x`; для чисел с плавающей точкой и комплексных чисел `+x` равно просто `x`, а `-x` является значением `x` с противоположным знаком.

Go также предоставляет следующие побитовые бинарные операторы, первые четыре из которых рассматривают свои операнды как битовые шаблоны, без понятий переноса или знака.

`&`	Побитовое И
`|`	Побитовое ИЛИ
`^`	Побитовое исключающее ИЛИ
`&^`	Сброс бита (И НЕ)
`<<`	Сдвиг влево
`>>`	Сдвиг вправо

Оператор `^` представляет собой побитовое исключающее ИЛИ (XOR) при использовании в качестве бинарного оператора, но при использовании в качестве унарного префиксного оператора он представляет собой побитовое отрицание, или дополнение; в таком случае оператор возвращает значение, в котором каждый бит равен инвертированному соответствующему биту его операнда. Оператор `&^` является оператором сброса бита: в выражении `z = x &^ y` каждый бит `z` равен `0`, если соответствующий бит `y` равен `1`; в противном случае он равен соответствующему биту `x`.

Приведенный ниже код показывает, как можно использовать битовые операции для интерпретации значения `uint8` в качестве компактного и эффективного набора восьми независимых битов. Он использует символы преобразования `%b` в `Printf` для вывода двоичных цифр числа; `08` уточняет поведение `%b`, заставляя дополнять результат нулями так, чтобы выводилось ровно 8 цифр.

```
var x uint8 = 1<<1 | 1<<5
var y uint8 = 1<<1 | 1<<2

fmt.Printf("%08b\n", x)    // "00100010", множество {1,5}
fmt.Printf("%08b\n", y)    // "00000110", множество {1,2}
fmt.Printf("%08b\n", x&y)  // "00000010", пересечение {1}
fmt.Printf("%08b\n", x|y)  // "00100110", объединение {1,2,5}
fmt.Printf("%08b\n", x^y)  // "00100100", симметричная разность {2,5}
fmt.Printf("%08b\n", x&^y) // "00100000", разность {5}

for i := uint(0); i < 8; i++ {
    if x&(1<<i) != 0 {     // Проверка принадлежности множеству
        fmt.Println(i)     // "1", "5"
    }
}
fmt.Printf("%08b\n", x<<1) // "01000100", множество {2,6}
fmt.Printf("%08b\n", x>>1) // "00010001", множество {0,4}
```

(В разделе 6.5 показана реализация целочисленных множеств, которые могут быть значительно больше байта.)

В операциях сдвига `x<<n` и `x>>n` операнд `n` определяет количество позиций сдвига и должен быть беззнаковым значением; операнд `x` может быть как беззнаковым,

так и знаковым. Арифметически сдвиг влево `x<<n` эквивалентен умножению на 2^n, а сдвиг вправо `x>>n` — делению на 2^n.

Сдвиг влево заполняет освобождающиеся биты нулями, так же, как и сдвиг вправо беззнакового значения; но сдвиг вправо знаковых чисел заполняет освобождаемые биты копиями знакового бита. По этой причине при рассмотрении целых чисел в качестве битового шаблона важно использовать беззнаковую арифметику.

Хотя Go предоставляет беззнаковые числа и арифметику, мы склонны использовать знаковый тип `int` даже для чисел, которые не могут быть отрицательными, например для таких, как длина массива, хотя `uint` может показаться более очевидным выбором. В действительности встроенная функция `len` возвращает знаковый `int`, как в приведенном цикле, который объявляет призовые медали в обратном порядке:

```
medals := []string{"золото", "серебро", "бронза"}
for i := len(medals) - 1; i >= 0; i-- {
    fmt.Println(medals[i]) // "бронза", "серебро", "золото"
}
```

Альтернатива оказалась бы катастрофической. Если бы `len` возвращала беззнаковое число, то переменная `i` также имела бы тип `uint`, и условие `i >= 0` было бы всегда верно по определению. После третьей итерации, в которой `i == 0`, инструкция `i--` привела бы к тому, что значение `i` стало бы равным не `-1`, а максимальному значению типа `uint` (например, $2^{64}-1$), и вычисление `medals[i]` при попытке получить доступ к элементу за пределами границ среза привело бы к сбою времени выполнения, или к *аварийной ситуации* (раздел 5.9).

По этой причине беззнаковые числа, как правило, используются только тогда, когда в выражении используются побитовые или необычные арифметические операторы, как, например, при реализации битовых множеств, при проведении анализа форматов бинарных файлов или для хеширования и криптографии. Как обычные неотрицательные значения они, как правило, не используются.

В общем случае для преобразования значения из одного типа в другой требуется явное преобразование типа, а бинарные арифметические и логические операторы (кроме сдвигов) должны иметь операнды одного и того же типа. Хотя это правило иногда и приводит к более длинным выражениям, зато при этом устраняется целый класс проблем и облегчается понимание программ.

В качестве примера, знакомого из других контекстов, рассмотрим последовательность

```
var apples int32 = 1
var oranges int16 = 2
var compote int = apples + oranges // Ошибка компиляции
```

Попытка компиляции этих трех объявлений приводит к сообщению об ошибке:

```
invalid operation: apples+oranges (mismatched types int32 and int16)

неверная операция: apples+oranges (несоответствие типов int32 и int16)
```

Это несоответствие типов можно разрешить несколькими способами, наиболее простой из которых — явное приведение к одному типу:

```
var compote = int(apples) + int(oranges)
```

Как описано в разделе 2.5, для каждого типа `T` операция преобразования `T(x)` преобразует значение `x` в тип `T`, если такое преобразование разрешено. Многие преобразования целого значения в целое не влекут за собой каких-либо изменений значения; они просто сообщают компилятору, как тому следует интерпретировать данное значение. Однако преобразование, которое сужает целое число большего размера до меньшего, или преобразование целого числа в число с плавающей точкой плавающей запятой и обратно может изменить значение или привести к потере точности:

```
f := 3.141          // float64
i := int(f)
fmt.Println(f, i)   // "3.141 3"
f = 1.99
fmt.Println(int(f)) // "1"
```

Преобразование значения с плавающей точкой в целое число приводит к отбрасыванию дробной части, т.е. к усечению по направлению к нулю. Следует избегать преобразований, в которых операнд находится вне диапазона целевого типа, потому что при этом поведение зависит от реализации:

```
f := 1e100    // float64
i := int(f)   // Результат зависит от реализации
```

Целочисленные литералы любого размера и типа могут быть записаны как обычные десятичные числа, как восьмеричные числа, если они начинаются с 0 (как `0666`), или как шестнадцатеричные, если они начинаются с `0x` или `0X` (как `0xdeadbeef`). Шестнадцатеричные цифры могут быть как прописными, так и строчными. Восьмеричные числа в настоящее время, похоже, используются для единственной цели — описания прав доступа к файлам в системах POSIX, зато шестнадцатеричные числа широко используются для того, чтобы подчеркнуть, что используется не числовое значение, а битовый шаблон.

При выводе чисел с использованием пакета `fmt` мы можем управлять системой счисления и форматом вывода с помощью символов преобразования `%d`, `%o` и `%x`, как показано в следующем примере:

```
o := 0666
fmt.Printf("%d %[1]o %#[1]o\n", o) // "438 666 0666"
x := int64(0xdeadbeef)
fmt.Printf("%d %[1]x %#[1]x %#[1]X\n", x)
// Вывод:
// 3735928559 deadbeef 0xdeadbeef 0XDEADBEEF
```

Обратите внимание на использование двух трюков `fmt`. Обычно форматная строка `Printf` содержит несколько символов преобразования `%`, которые требуют того же количества дополнительных аргументов, но `[1]` после `%` говорит функции `Printf` о

том, что ей следует использовать первый операнд снова и снова. Во-вторых, наличие # при символах преобразования %o, %x или %X говорит функции Printf о том, что при выводе должен быть добавлен префикс, указывающий систему счисления — 0, 0x или 0X соответственно.

Литералы записываются как символ в одинарных кавычках. Простейший пример с использованием ASCII-символа — 'a', но так можно записать любой символ Unicode — либо непосредственно, либо с помощью числовых управляющих последовательностей, как мы вскоре увидим.

Руны выводятся с помощью символов преобразования %c, или %q — если требуются кавычки:

```
ascii   := 'a'
unicode := '★'
newline := '\n'
fmt.Printf("%d %[1]c %[1]q\n", ascii)   // "97 a 'a'"
fmt.Printf("%d %[1]c %[1]q\n", unicode) // "9733 ★ '★'"
fmt.Printf("%d %[1]q\n", newline)       // "10 '\n'"
```

3.2. Числа с плавающей точкой

Go предоставляет два варианта чисел с плавающей точкой разного размера, float32 и float64. Их арифметические свойства регулируются стандартом IEEE 754 и реализованы на всех современные процессорах.

Значения этих числовых типов находятся в диапазоне от самых маленьких до огромных. Предельные значения с плавающей точкой можно найти в пакете math. Константа math.MaxFloat32, наибольшее значение типа float32, примерно равна 3.4e38, а math.MaxFloat64 — примерно 1.8e308. Наименьшие положительные значения составляют около 1.4e45 и 4.9e324 соответственно.

Тип float32 обеспечивает приблизительно шесть десятичных цифр точности, в то время как точность, обеспечиваемая типом float64, составляет около 15 цифр; тип float64 в большинстве случаев следует предпочитать типу float32, поскольку при использовании последнего, если не принимать специальные меры, быстро накапливается ошибка, а наименьшее положительное целое число, которое не может быть представлено типом float32, не очень велико:

```
var f float32 = 16777216  // 1 << 24
fmt.Println(f == f+1)     // "true"!
```

Числа с плавающей точкой могут быть записаны буквально, с использованием десятичной записи:

```
const e = 2.71828     // (Приблизительно)
```

В записи могут быть опущены цифры перед десятичной точкой (.707) и после нее (1.). Очень малые и очень большие числа лучше записывать с использованием научного формата, с буквой e или E перед десятичной степенью:

```
const Avogadro = 6.02214129e23
const Planck   = 6.62606957e34
```

Значения с плавающей точкой удобно выводить с помощью символов преобразования `%g` функции `Printf`, которые выбирают наиболее компактное представление значения с надлежащей точностью, но для таблиц данных могут оказаться предпочтительнее символы преобразования `%e` (с показателем степени) или `%f` (без показателя степени). Все три варианта символов преобразования позволяют указывать ширину поля и числовую точность:

```
for x := 0; x < 8; x++ {
    fmt.Printf("x = %d e^x = %8.3f\n", x, math.Exp(float64(x)))
}
```

Приведенный выше исходный текст выводит степени *e* с тремя десятичными цифрами точности, выровненными в поле длиной 8 символов:

```
x = 0 e^x =    1.000
x = 1 e^x =    2.718
x = 2 e^x =    7.389
x = 3 e^x =   20.086
x = 4 e^x =   54.598
x = 5 e^x =  148.413
x = 6 e^x =  403.429
x = 7 e^x = 1096.633
```

В дополнение к большой коллекции математических функций в пакете `math` имеются функции для создания и обнаружения специальных значений, определенных стандартом IEEE 754: положительной и отрицательной бесконечности (которые представляют числа чрезмерной величины и результат деления на нуль) и NaN ("not a number" — не число), являющееся результатом таких математически сомнительных операций, как `0/0` или `Sqrt(-1)`:

```
var z float64
fmt.Println(z, -z, 1/z, -1/z, z/z) // "0 -0 +Inf -Inf NaN"
```

Функция `math.IsNaN` проверяет, является ли ее значением NaN, а `math.NaN` возвращает это значение. Имеется соблазн использовать NaN в качестве значения-ограничителя в числовых вычислениях, но проверка, равен ли результат конкретных вычислений NaN, достаточно опасна, поскольку любое сравнение с NaN *всегда* дает значение `false`:

```
nan := math.NaN()
fmt.Println(nan == nan, nan < nan, nan > nan) // "false false false"
```

Если функция, которая возвращает в качестве результата значение с плавающей точкой, может привести к ошибке или сбою, об этом лучше сообщать отдельным возвращаемым значением, как показано ниже:

```
func compute() (value float64, ok bool) {
    // ...
```

```
    if failed {
        return 0, false
    }
    return result, true
}
```

Приведенная далее программа иллюстрирует графические вычисления с плавающей точкой. Она строит график функции от двух переменных `z = f(x,y)` в виде трехмерной сетки с помощью SVG (Scalable Vector Graphics — масштабируемая векторная графика), стандартного XML-формата для черчения линий. На рис. 3.1 показан пример вывода программы для функции `sin(r)/r`, где `r` представляет собой `sqrt(x*x+y*y)`.

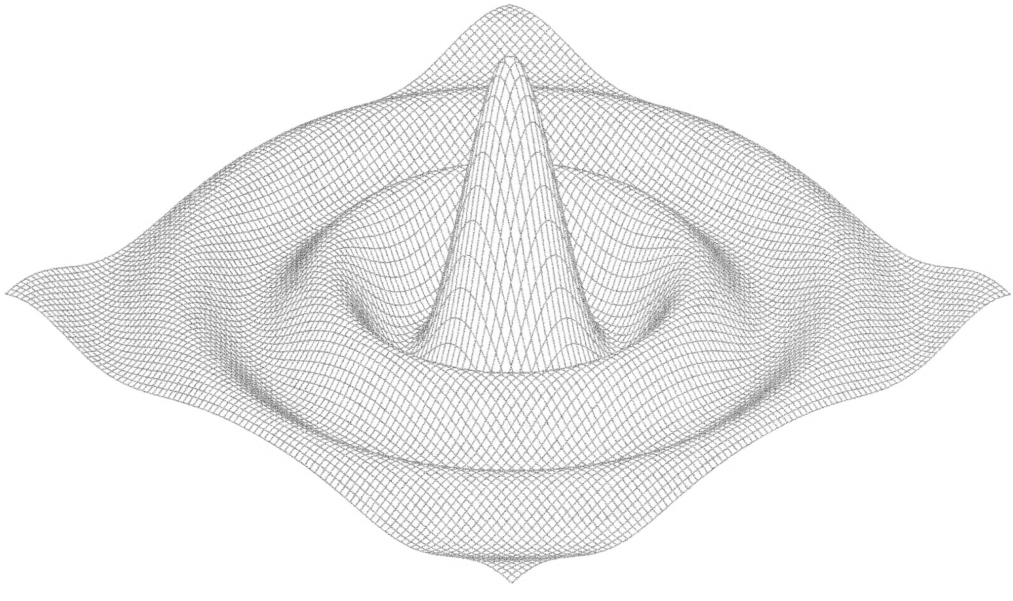


Рис. 3.1. Чертеж поверхности функции `sin(r)/r`

aopl.io/ch3/surface

```
// Surface вычисляет SVG-представление трехмерного графика функции.
package main

import (
    "fmt"
    "math"
)

const (
    width, height = 600, 320        // Размер канвы в пикселях
    cells         = 100             // Количество ячеек сетки
    xyrange       = 30.0            // Диапазон осей
```

```
                                    // (-xyrange..+ xyrange)
    xyscale       = width/2/xyrange // Пикселей в единице x или y
    zscale        = height * 0.4    // Пикселей в единице z
    angle         = math.Pi / 6     // Углы осей x, y (=30°)
)

var sin30, cos30 = math.Sin(angle),math.Cos(angle) //sin(30°),cos(30°)

func main() {
    fmt.Printf("<svg xmlns='http://www.w3.org/2000/svg' "+
        "style='stroke: grey; fill: white; stroke-width: 0.7' "+
        "width='%d' height='%d'>", width, height)
    for i := 0; i < cells; i++ {
        for j := 0; j < cells; j++ {
            ax, ay := corner(i+1, j)
            bx, by := corner(i, j)
            cx, cy := corner(i, j+1)
            dx, dy := corner(i+1, j+1)
            fmt.Printf("<polygon points='%g,%g %g,%g %g,%g %g,%g'/>\n",
                ax, ay, bx, by, cx, cy, dx, dy)
        }
    }
    fmt.Println("</svg>")
}

func corner(i, j int) (float64, float64) {
    // Ищем угловую точку (x,y) ячейки (i,j).
    x := xyrange * (float64(i)/cells - 0.5)
    y := xyrange * (float64(j)/cells - 0.5)
    // Вычисляем высоту поверхности z
    z := f(x, y)
    // Изометрически проецируем (x,y,z) на двумерную канву SVG (sx,sy)
    sx := width/2 + (xy)*cos30*xyscale
    sy := height/2 + (x+y)*sin30*xyscale - z*zscale
    return sx, sy
}

func f(x, y float64) float64 {
    r := math.Hypot(x, y) // Расстояние от (0,0)
    return math.Sin(r) / r
}
```

Обратите внимание, что функция `corner` возвращает два значения, координаты угла ячейки.

Для объяснения работы программы нужно знать только основы геометрии, но мы можем просто пропустить его, поскольку наша цель — всего лишь проиллюстрировать вычисления с плавающей точкой. Суть программы состоит в выполнении отображения между тремя различными системами координат, показанными на рис. 3.2.

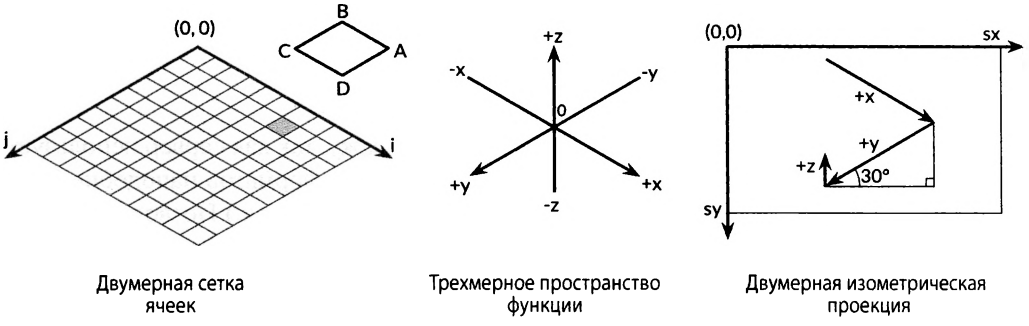


Рис. 3.2. Три различные системы координат

Первая — это двумерная сетка размером 100×100 ячеек, идентифицируемых значениями целочисленных координат (i, j), начиная с (0, 0) в дальнем углу. Мы выполняем черчение от дальнего конца к ближнему, так что дальние многоугольники могут быть скрыты находящимися ближе.

Вторая система координат представляет собой сетку трехмерных координат (x, y, z) с плавающей точкой, где x и y являются линейными функциями от i и j, перемещенные так, что начало координат находится в центре, и масштабированные с помощью константы `xyrange`. Высота z представляет собой значение функции поверхности $f(x, y)$.

Третья система координат представляет собой канву двумерного изображения с точкой (0, 0) в левом верхнем углу. Точки на этой плоскости обозначается как (sx, sy). Мы используем изометрическую проекцию для отображения каждой трехмерной точки (x, y, z) на двумерную канву. Чем дальше справа на канве находится точка, тем больше ее значение x или *меньше* ее значение y. А чем ниже на канве находится точка, тем больше ее значение x *или* значение y и меньше значение z. Вертикальные и горизонтальные масштабные множители для x и y вычисляются как синус и косинус угла 30°. Масштабный множитель для z, равный 0.4, выбран произвольно.

Для каждой ячейки в двумерной сетке функция `main` вычисляет координаты на канве четырех углов многоугольника $ABCD$, где вершина B соответствует (i, j), а вершины A, C и D являются ее соседями, а затем выводит SVG-команду его черчения.

Упражнение 3.1. Если функция `f` возвращает значение `float64`, не являющееся конечным, SVG-файл содержит неверные элементы `<polygon>` (хотя многие визуализаторы SVG успешно обрабатывают эту ситуацию). Измените программу так, чтобы некорректные многоугольники были опущены.

Упражнение 3.2. Поэкспериментируйте с визуализациями других функций из пакета `math`. Сможете ли вы получить изображения наподобие коробки для яиц, седла или холма?

Упражнение 3.3. Окрасьте каждый многоугольник цветом, зависящим от его высоты, так, чтобы пики были красными (`#ff0000`), а низины — синими (`#0000ff`).

Упражнение 3.4. Следуя подходу, использованному в примере с фигурами Лиссажу из раздела 1.7, создайте веб-сервер, который вычисляет поверхности и возвращает

клиенту SVG-данные. Сервер должен использовать в ответе заголовок `ContentType` наподобие следующего:

```
w.Header().Set("ContentType", "image/svg+xml")
```

(Этот шаг не был нужен в примере с фигурами Лиссажу, так как сервер использует стандартную эвристику распознавания распространенных форматов наподобие PNG по первым 512 байтам ответа и генерирует корректный заголовок.) Позвольте клиенту указывать разные параметры, такие как высота, ширина и цвет, в запросе HTTP.

3.3. Комплексные числа

Go предоставляет комплексные числа двух размеров, `complex64` и `complex128`, компонентами которых являются `float32` и `float64` соответственно. Встроенная функция `complex` создает комплексное число из действительной и мнимой компонент, а встроенные функции `real` и `imag` извлекают эти компоненты:

```
var x complex128 = complex(1, 2) // 1+2i
var y complex128 = complex(3, 4) // 3+4i
fmt.Println(x*y)                 // "(5+10i)"
fmt.Println(real(x*y))           // "5"
fmt.Println(imag(x*y))           // "10"
```

Если непосредственно за литералом с плавающей точкой или за десятичным целочисленным литералом следует `i`, например `3.141592i` или `2i`, такой литерал становится *мнимым литералом*, обозначающим комплексное число с нулевым действительным компонентом:

```
fmt.Println(1i * 1i) // "(-1+0i)", i² = -1
```

Согласно правилам константной арифметики комплексные константы могут быть прибавлены к другим константам (целочисленным или с плавающей точкой, действительным или мнимым), позволяя нам записывать комплексные числа естественным образом, как, например, `1 + 2i` или, что то же самое, `2i + 1`. Показанные выше объявления `x` и `y` могут быть упрощены:

```
x := 1 + 2i
y := 3 + 4i
```

Комплексные числа можно проверять на равенство с помощью операторов `==` и `!=`. Два комплексных числа равны тогда и только тогда, когда равны их действительные части и их мнимые части.

Пакет `math/cmplx` предоставляет библиотечные функции для работы с комплексными числами, такие как комплексный квадратный корень или возведение в степень:

```
fmt.Println(cmplx.Sqrt(-1)) // "(0+1i)"
```

Приведенная далее программа использует арифметику `complex128` для генерации множества Мандельброта.

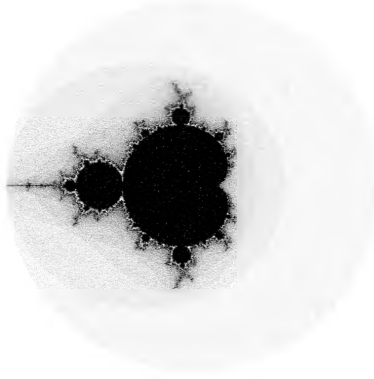


Рис. 3.3. Множество Мандельброта

gopl.io/ch3/mandelbrot

```
// Mandelbrot создает PNG-изображение фрактала Мандельброта.
package main

import (
    "image"
    "image/color"
    "image/png"
    "math/cmplx"
    "os"
)

func main() {
    const (
        xmin, ymin, xmax, ymax = -2, -2, +2, +2
        width, height          = 1024, 1024
    )
    img := image.NewRGBA(image.Rect(0, 0, width, height))
    for py := 0; py < height; py++ {
        y := float64(py) / height * (ymax - ymin)
        +ymin
        for px := 0; px < width; px++ {
            x := float64(px)/width*(xmax-xmin) + xmin
            z := complex(x, y)
            // Точка (px, py) представляет комплексное значение z.
            img.Set(px, py, mandelbrot(z))
        }
    }
    png.Encode(os.Stdout, img) // Примечание: игнорируем ошибки
}

func mandelbrot(z complex128) color.Color {
    const iterations = 200
    const contrast = 15
```

```
    var v complex128
    for n := uint8(0); n < iterations; n++ {
        v = v*v + z
        if cmplx.Abs(v) > 2 {
            return color.Gray{255 - contrast*n}
        }
    }
    return color.Black
}
```

Два вложенных цикла проходят по всем точкам растрового изображения размером 1024×1024 в оттенках серого цвета, представляющего часть комплексной плоскости от –2 до +2. Программа проверяет, позволяет ли многократное возведение в квадрат и добавление числа, представляющего точку, "сбежать" из круга радиусом 2. Если позволяет, то данная точка закрашивается оттенком, соответствующим количеству итераций, потребовавшихся для "побега". Если не позволяет, данное значение принадлежит множеству Мандельброта, и точка остается черной. Наконец программа записывает в стандартный поток вывода изображение в PNG-кодировке, показанное на рис. 3.3.

Упражнение 3.5. Реализуйте полноцветное множество Мандельброта с использованием функции `image.NewRGBA` и типа `color.RGBA` или `color.YCbCr`.

Упражнение 3.6. Супервыборка (supersampling) — это способ уменьшить эффект пикселизации путем вычисления значений цвета в нескольких точках в пределах каждого пикселя и их усреднения. Проще всего разделить каждый пиксель на четыре "подпикселя". Реализуйте описанный метод.

Упражнение 3.7. Еще один простой фрактал использует метод Ньютона для поиска комплексных решений уравнения $z^4-1 = 0$. Закрасьте каждую точку цветом, соответствующим тому корню из четырех, которого она достигает, а интенсивность цвета должна соответствовать количеству итераций, необходимых для приближения к этому корню.

Упражнение 3.8. Визуализация фракталов при высоком разрешении требует высокой арифметической точности. Реализуйте один и тот же фрактал с помощью четырех различных представлений чисел: `complex64`, `complex128`, `big.Float` и `big.Rat`. (Два последних типа вы найдете в пакете `math/big`. `Float` использует произвольную, но ограниченную точность для чисел с плавающей точкой; `Rat` обеспечивает неограниченную точность для рациональных чисел.) Сравните производительность и потребление памяти при использовании разных типов. При каком уровне масштабирования артефакты визуализации становятся видимыми?

Упражнение 3.9. Напишите программу веб-сервера, который визуализирует фракталы и выводит данные изображения клиенту. Позвольте клиенту указывать значения x, y и масштабирования в качестве параметров HTTP-запроса.

3.4. Булевы значения

Тип `bool`, или *булев тип* (boolean), имеет только два возможных значения — `true` и `false`. Булевыми являются условия в инструкциях `if` и `for`, а операторы

сравнения наподобие == и < дают булев результат. Унарный оператор ! представляет собой логическое отрицание, так что `!true` равно `false` (можно сказать, что `(!true==false)==true`, хотя в качестве хорошего стиля мы всегда упрощаем излишне сложные булевы выражения наподобие упрощения `x==true` до `x`).

Булевы значения могут быть объединены с помощью операторов && (И) и || (ИЛИ), при вычислении которых используется *сокращенное* (shortcircuit) вычисление: если ответ определяется значением левого операнда, правый операнд не вычисляется. Это делает безопасными выражения, подобные следующему:

```
s != "" && s[0] == 'x'
```

Выражение `s[0]` приводит к сбою программы при применении к пустой строке.

Поскольку оператор && имеет более высокий приоритет, чем оператор || (мнемоника: && представляет собой булево умножение, а || — булево сложение), для условий приведенного ниже вида не требуются никакие скобки:

```
if 'a' <= c && c <= 'z' ||
    'A' <= c && c <= 'Z' ||
    '0' <= c && c <= '9' {
    // ...ASCII-буква или цифра...
}
```

Не существует неявного преобразования булева значения в числовое значение наподобие 0 или 1 или обратно. Необходимо использовать явную инструкцию `if`, как показано ниже:

```
i := 0
if b {
    i = 1
}
```

При частой необходимости такой операции может иметь смысл написание специальной функции преобразования:

```
// btoi возвращает 1, если b равно true, и 0, если false.
func btoi(b bool) int {
    if b {
        return 1
    }
    return 0
}
```

Обратная операция настолько проста, что она не стоит написания отдельной функции, но для симметрии мы приведем ее здесь:

```
// itob указывает, имеет ли i ненулевое значение.
func itob(i int) bool { return i != 0 }
```

3.5. Строки

Строка представляет собой неизменяемую последовательность байтов. Строки могут содержать произвольные данные, в том числе байты со значением 0, но обычно они содержат удобочитаемый для человека текст. Традиционно текстовые строки интерпретируются как последовательности символов Unicode в кодировке UTF-8, о чем подробнее мы поговорим ниже.

Встроенная функция `len` возвращает количество байтов (не символов!) в строке, а операция *индексирования* `s[i]` возвращает *i*-й байт строки `s`, где $0 \le i < \texttt{len(s)}$:

```
s := "hello, world"
fmt.Println(len(s))     // "12"
fmt.Println(s[0], s[7]) // "104 119" ('h' and 'w')
```

Попытка обращения к байту вне этого диапазона приводит к аварийной ситуации:

```
c := s[len(s)] // Авария: индекс вне диапазона
```

i-й байт строки не обязательно является *i*-м *символом* строки, поскольку кодировка UTF-8 символов, не являющихся символами ASCII, требует двух или более байтов. Работа с символами будет вкратце описана ниже.

Операция получения *подстроки* `s[i:j]` дает новую строку, состоящую из байтов исходной строки, начиная с индекса `i` и до (но не включая) байта с индексом `j`. Результирующая строка содержит `j-i` байтов.

```
fmt.Println(s[0:5]) // "hello"
```

При выходе индексов за пределы строки или если `j` меньше `i` возникает аварийная ситуация.

Каждый из операндов `i` и `j` может быть опущен, и в этом случае используются значения по умолчанию — соответственно `0` (начало строки) и `len(s)` (ее конец).

```
fmt.Println(s[:5]) // "hello"
fmt.Println(s[7:]) // "world"
fmt.Println(s[:])  // "hello, world"
```

Оператор + создает новую строку путем конкатенации двух строк:

```
fmt.Println("goodbye" + s[5:]) // "goodbye, world"
```

Строки можно сравнивать с помощью операторов сравнения наподобие == и <; сравнение выполняется байт за байтом, поэтому оно происходит в естественном лексикографическом порядке.

Строковые значения являются неизменяемыми: последовательность байтов, содержащихся в строковом значении, не может быть изменена, хотя, конечно же, мы можем присвоить строковой переменной новое значение. Например, для добавления одной строки к другой мы можем написать

```
s := "левая нога"
t := s
s += ", правая нога"
```

Этот код не изменяет строку, изначально хранившуюся в s, но приводит к тому, что в s теперь хранится новая строка, образованная инструкцией +=; в то же время в t хранится старая строка.

```
fmt.Println(s) // "левая нога, правая нога"
fmt.Println(t) // "левая нога"
```

Поскольку строки неизменяемы, конструкции, которые пытаются изменить данные строки, не допускаются:

```
s[0] = 'L' // Ошибка компиляции: нельзя присваивать s[0]
```

Неизменяемость означает, что две копии строки могут вполне безопасно разделять одну и ту же память, что делает копирование строки любой длины очень дешевой операцией. Аналогично строка s и подстроки, такие как s[7:], могут безопасно разделять одни и те же данные, поэтому операция получения подстроки также очень дешевая. В обоих случаях не выделяется никакая новая память. На рис. 3.4 показано расположение строки и двух ее подстрок, использующих один и тот же массив байтов.

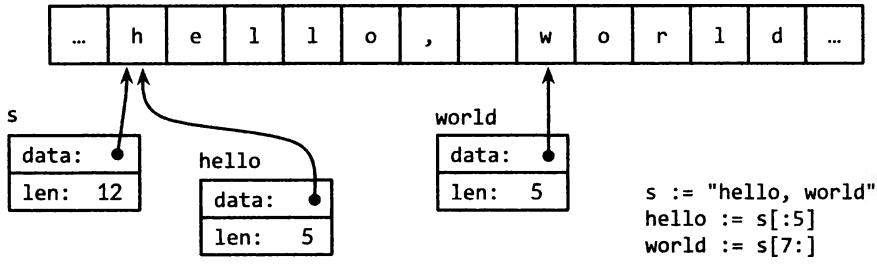


Рис. 3.4. Строка и две подстроки

3.5.1. Строковые литералы

Строковое значение можно записать как *строковый литерал*, последовательность байтов, заключенную в двойные кавычки:

```
"Привет, 世界"
```

Поскольку исходные файлы Go всегда используют кодировку UTF-8, а по соглашению текстовые строки Go интерпретируются как закодированные с использованием UTF-8, мы можем включать символы Unicode в строковые литералы.

В строковых литералах в двойных кавычках для вставки произвольных значений байтов в строку могут использоваться *управляющие последовательности*, начинающиеся с обратной косой черты \. Одно множество управляющих последовательностей обрабатывает управляющие коды ASCII, такие как символы новой строки, возврат каретки и символы табуляции.

`\a`	Звуковой сигнал
`\b`	Забой (backspace)
`\f`	Перевод страницы
`\n`	Новая строка
`\r`	Возврат каретки
`\t`	Табуляция
`\v`	Вертикальная табуляция
`\'`	Одинарная кавычка (только в символьном литерале `'\''`)
`\"`	Двойные кавычки (только в литералах `"..."`)
`\\`	Обратная косая черта

С помощью управляющих последовательностей в строковые литералы могут быть включены также произвольные байты. *Шестнадцатеричные управляющие последовательности* записываются как `\x`*hh*, ровно с двумя шестнадцатеричными цифрами *h* (в верхнем или нижнем регистре). *Восьмеричная управляющая последовательность* записывается как `\`*ooo* с ровно тремя восьмеричными цифрами (от 0 до 7), не превышающими значение `\377`. Обе они обозначают один байт с указанным значением. Позже вы узнаете, как численно записывать в строковых литералах символы Unicode.

Неформатированный строковый литерал (raw string literal) записывается с помощью обратных одинарных кавычек (`` `...` ``) вместо двойных. Внутри неформатированного строкового литерала управляющие последовательности не обрабатываются; содержимое принимается буквально, включая обратные косые черты и символы новой строки, так что неформатированный строковый литерал может состоять из нескольких строк исходного текста. Единственная обработка состоит в удалении символов возврата каретки, чтобы значение строки было одинаковым на всех платформах, в том числе на тех, на которых в текстовые файлы традиционно включаются эти символы возврата каретки.

Использование неформатированных строковых литералов — это удобный способ записи регулярных выражений, в которых обычно очень много обратных косых черт. Они также полезны для шаблонов HTML, литералов JSON, сообщений об использовании программы и тому подобного, что часто занимает несколько строк текста.

```
const GoUsage = `Go - инструмент для работы с исходным текстом Go.
Использование:
go команда [аргументы]
...`
```

3.5.2. Unicode

Давным-давно жизнь была простой и в ней хватало — по крайней мере в США — только одного небольшого набора символов ASCII (American Standard Code for Information Interchange — Американский стандартный код для обмена информацией). ASCII, или, точнее, US-ASCII, использует 7 бит для представления 128 “символов”:

прописных и строчных букв английского алфавита, цифр и разнообразных знаков пунктуации и управляющих символов устройств. Для большинства задач тех времен этого было достаточно, но бо́льшая часть населения мира при этом была лишена возможности пользоваться собственными системами письменности в компьютерах. С ростом Интернета данные на многочисленных языках стали гораздо более распространенным явлением. Каким же образом можно работать со всем этим богатым материалом, причем работать по возможности эффективно?

Ответом на этот вопрос является *Unicode* (`unicode.org`), в котором собраны все символы из всех мировых систем письменности плюс разнообразные символы ударений и прочие диакритические знаки, управляющие коды наподобие символов табуляции и возврата каретки и многое другое. Каждому такому символу назначен стандартный номер — *код символа Unicode* (Unicode code point), или, в терминологии Go, *руна* (rune).

Unicode версии 8 определяет коды для более чем 120 тысяч символов более чем в сотне языков и алфавитов. Как же они представляются в компьютерных программах и данных? Естественным типом данных для хранения отдельной руны является тип `int32`, и именно он используется в Go; для этого в Go используется синоним типа `rune`.

Мы могли бы представлять последовательность рун как последовательность значений типа `int32`. В таком представлении, которое называется UTF-32 или UCS-4, каждый код символа имеет один и тот же размер — 32 бита. Это простое и универсальное кодирование использует гораздо больше памяти, чем необходимо, поскольку большинство текстов в компьютерах используют только символы из кодировки ASCII, которой требуется только 8 битов, или 1 байт, на символ. Практически все сколь-нибудь широко используемые символы имеют коды меньше 65536, вписывающиеся в 16 битов. Не можем ли мы найти лучшее решение?

3.5.3. UTF-8

UTF-8 представляет собой кодировку переменной длины символов Unicode в виде байтов. Кодировка UTF-8 была изобретена двумя из создателей Go, Кеном Томпсоном (Ken Thompson) и Робом Пайком (Rob Pike), и в настоящее время является стандартом Unicode. Она использует от 1 до 4 байтов для представления каждой руны, но при этом только 1 байт для символов ASCII и только 2 или 3 байта для большинства распространенных рун. Старшие биты первого байта кодировки руны указывают, сколько байтов следуют за первым. Нулевой старший бит указывает, что это 7-битовый символ ASCII, в котором каждая руна занимает только 1 байт (а потому он идентичен обычному ASCII). Старшие биты `110` указывают, что руна занимает 2 байта; второй байт при этом начинается с битов `10`. Бо́льшие руны имеют аналогичные кодировки.

`0xxxxxxx`	Руны 0–127	(ASCII)
`110xxxxx 10xxxxxx`	128–2047	(Значения <128 не используются)
`1110xxxx 10xxxxxx 10xxxxxx`	2048–65535	(Значения <2048 не используются)
`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`	65536–0x10ffff	(Прочие значения не используются)

Переменная длина кодировки исключает возможность прямой индексации для доступа к *n*-му символу строки, но UTF-8 имеет множество полезных свойств, компенсирующих этот недостаток. Кодировка компактна, совместима с ASCII и обладает свойством самосинхронизации: всегда можно найти начало символа, просмотрев не более чем три байта. Этот код является префиксным, так что его можно декодировать слева направо без каких-либо неоднозначностей или предпросмотра. Никакой код руны не является подстрокой другого кода или даже последовательности других кодов, так что вы можете искать руны с помощью простого поиска составляющих ее байтов, не беспокоясь о контексте. Лексикографический порядок байтов соответствует порядку кода Unicode, так что сортировка UTF-8 работает естественным образом. Не существует встроенных нулевых байтов, что удобно для языков программирования, в которых используются нулевые символы для завершения строк.

Исходные файлы Go всегда закодированы с использованием кодировки UTF-8, и эта кодировка является предпочтительной для текстовых строк, с которыми работают программы Go. Пакет `unicode` предоставляет функции для работы с отдельными рунами (например, чтобы отличать буквы от цифр или преобразовывать прописные буквы в строчные), а пакет `unicode/utf8` предоставляет функции для кодирования и декодирования рун в виде байтов с использованием UTF-8.

Многие символы Unicode трудно набирать на клавиатуре или визуально отличить от других, визуально схожих; некоторые из них просто невидимы. Управляющие последовательности Unicode в строковых литералах Go позволяют указывать символы Unicode с помощью их числового кода. Существуют две разновидности управляющих последовательностей — `\u`*`hhhh`* для 16-разрядных значений и `\U`*`hhhhhhhh`* для 32-разрядных, где каждое *`h`* является шестнадцатеричной цифрой. Необходимость в 32-разрядном режиме возникает очень редко. Каждая из управляющих последовательностей представляет собой кодировку UTF-8 указанного символа Unicode. Таким образом, например, все приведенные далее строковые литералы представляют одну и ту же шестибайтовую строку:

```
"世界"
"\xe4\xb8\x96\xe7\x95\x8c"
"\u4e16\u754c"
"\U00004e16\U0000754c"
```

Три приведенные выше строки с управляющими последовательностями представляют собой альтернативную запись первой строки, но значения, которые они описывают, одинаковы.

Управляющие последовательности Unicode могут использоваться и для литералов рун. Приведенные далее три литерала эквивалентны:

```
'世'    '\u4e16'    '\U00004e16'
```

Руна, значение которой меньше 256, может быть записана с помощью одной шестнадцатеричной управляющей последовательности, такой как `'\x41'` для `'A'`, но для более высоких значений следует использовать управляющие последовательности `\u` или `\U`. Следовательно, `'\xe4\xb8\x96'` не является корректным литералом руны,

несмотря на то что эти три байта являются корректной кодировкой UTF-8 одного символа Unicode.

Благодаря указанным выше свойствам UTF-8 многие строковые операции не требуют декодирования. Мы можем проверить, не содержит ли одна строка другую в качестве префикса:

```
func HasPrefix(s, prefix string) bool {
    return len(s) >= len(prefix) && s[:len(prefix)] == prefix
}
```

или в качестве суффикса:

```
func HasSuffix(s, suffix string) bool {
    return len(s) >= len(suffix) && s[len(s)-len(suffix):] == suffix
}
```

или в качестве подстроки:

```
func Contains(s, substr string) bool {
    for i := 0; i < len(s); i++ {
        if HasPrefix(s[i:], substr) {
            return true
        }
    }
    return false
}
```

используя для закодированного с помощью кодировки UTF-8 текста ту же логику, что и для обычной последовательности байтов. Для других кодировок это было бы неверным решением. (Приведенные выше функции взяты из пакета `strings`, хотя реализация функции `Contains` в этом пакете для более эффективного поиска использует хеширование.)

С другой стороны, если нас действительно интересуют отдельные символы Unicode, то мы должны использовать другие механизмы. Рассмотрим строку, которая включает в себя два иероглифа. На рис. 3.5 показано ее представление в памяти. Строка содержит 13 байт, но если интерпретировать ее как UTF-8, то в ней закодировано только девять кодов Unicode, или рун:

```
import "unicode/utf8"

s := "Hello, 世界"
fmt.Println(len(s))                    // "13"
fmt.Println(utf8.RuneCountInString(s)) //  "9"
```

Для работы с этими символами необходимо декодирование UTF-8. Пакет `unicode/utf8` предоставляет декодер, который мы можем использовать следующим образом:

```
for i := 0; i < len(s); {
    r, size := utf8.DecodeRuneInString(s[i:])
    fmt.Printf("%d\t%c\n", i, r)
```

```
    i += size
}
```

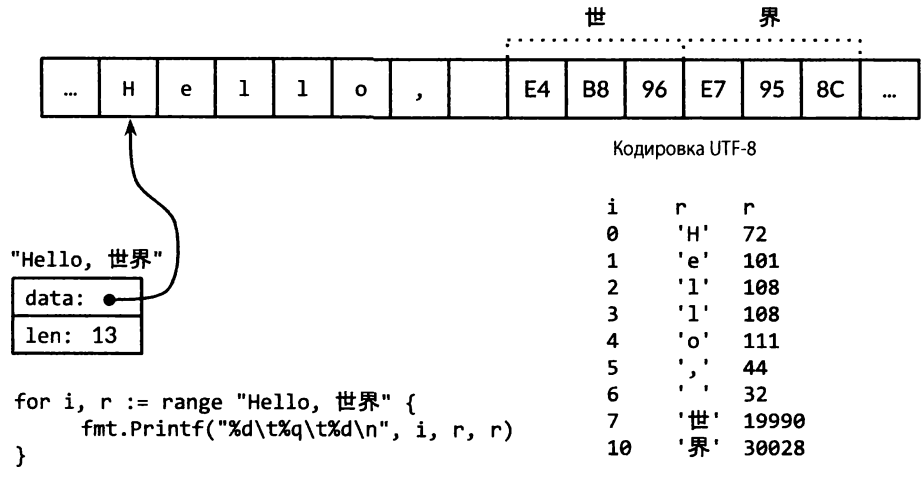


Рис. 3.5. Цикл по диапазону, декодирующий закодированную UTF-8 строку

Каждый вызов `DecodeRuneInString` возвращает `r` (саму руну) и `size`, количество байтов, занятых UTF-8 кодом `r`. Размер используется для обновления индекса байтов `i` для следующей руны в строке. Но это достаточно неуклюжее решение, и нам постоянно необходимы циклы такого рода. К счастью, цикл по диапазону Go, будучи примененным к строке, выполняет декодирование UTF-8 неявно. Вывод приведенного ниже цикла показан также на рис. 3.5; обратите внимание, как индекс перемещается более чем на один байт для каждой руны, не являющейся ASCII-символом.

```
for i, r := range "Hello, 世界" {
    fmt.Printf("%d\t%q\t%d\n", i, r, r)
}
```

Для подсчета количества рун в строке можно использовать простой цикл по `range`:

```
n := 0
for _, _ = range s {
    n++
}
```

Как и в прочих разновидностях цикла `range`, можно опустить ненужные нам переменные:

```
n := 0
for range s {
    n++
}
```

Для решения этой задачи можно также просто вызвать `utf8.RuneCountInString(s)`.

Ранее мы упоминали, что то, что тексты строк в Go интерпретируются как последовательности символов Unicode в кодировке UTF-8 в основном по соглашению, но для правильного использования циклов по диапазону это больше, чем соглашение, — это необходимость. Что произойдет, если мы применим такой цикл к строке, содержащей произвольные бинарные данные или, например, данные UTF-8, содержащие ошибки?

Декодер UTF-8, получая при явном вызове `utf8.DecodeRuneInString` или неявно в цикле по диапазону некорректный входной байт, генерирует специальный *замещающий символ* Unicode, `'\uFFFD'`, который обычно выглядит как белый вопросительный знак в черном ромбе (). Когда программа встречает руну с этим значением, это, как правило, означает, что некая часть системы, генерирующей строковые данные, была некорректна в трактовке закодированного текста.

Кодировка UTF-8 оказывается исключительно удобной в качестве формата обмена данными, но в пределах программы руны могут оказаться более удобными, поскольку имеют одинаковый размер и, таким образом, легко индексируются в массивах и срезах.

Преобразование `[]rune`, примененное к строке в кодировке UTF-8, возвращает последовательность символов Unicode, закодированную в этой строке:

```
// Слово "программа", записанное по-японски на катакане
s := "プログラム"
fmt.Printf("% x\n",s) //"e3 83 97 e3 83 ad e3 82 b0 e3 83 a9 e3 83 a0"
r := []rune(s)
fmt.Printf("%x\n",r)  //"[30d7 30ed 30b0 30e9 30e0]"
```

(Символы преобразования `% x` в первом вызове `Printf` вставляют пробелы между каждой парой шестнадцатеричных цифр.)

Если срез рун преобразуется в строку, генерируется конкатенация UTF-8 кодов для каждой руны:

```
fmt.Println(string(r)) // "プログラム"
```

Преобразование целочисленного значения в строку рассматривает это целочисленное значение как значение руны и дает представление этой руны в кодировке UTF-8:

```
fmt.Println(string(65))     // "A", но не "65"
fmt.Println(string(0x4eac)) // "C"
```

Если руна некорректна, вместо нее подставляется замещающий символ:

```
fmt.Println(string(1234567)) // ""
```

3.5.4. Строки и байтовые срезы

Для работы со строками в особенности важны четыре стандартных пакета: `bytes`, `strings`, `strconv` и `unicode`. Пакет `strings` предоставляет множество функций для поиска, замены, сравнения, обрезки, разделения и объединения строк.

Пакет `bytes` содержит аналогичные функции для работы со срезами байтов `[]byte`, который имеет некоторые свойства, общие со свойствами строк. Поскольку строки неизменяемы, инкрементное построение строк может включать огромное количество выделений памяти и копирований. В таких случаях более эффективным является использование типа `bytes.Buffer`, о котором мы вскоре поговорим.

Пакет `strconv` предоставляет функции для преобразования булевых значений, целых чисел и чисел с плавающей точкой в строковое представление (и обратно). Функции для классификации рун, такие как `IsDigit`, `IsLetter`, `IsUpper` и `IsLower`, можно найти в пакете `unicode`. Каждая из этих функций получает в качестве аргумента отдельную руну и возвращает булево значение. Функции преобразования наподобие `ToUpper` и `ToLower` преобразуют руну в руну в указанном регистре, если эта руна является буквой. Все эти функции используют категории стандарта Unicode для букв, цифр и т.д. В пакете `strings` имеются подобные функции, также именуемые `ToUpper` и `ToLower`, которые возвращают новые строки с указанными преобразованиями, примененными к каждому символу исходной строки.

Показанная ниже функция `basename` основана на утилите с тем же именем из Unix. В нашей версии `basename(s)` убирает из `s` все префиксы, имеющие вид пути файловой системы с компонентами, разделенными косыми чертами, а также суффикс, представляющий тип файла:

```
fmt.Println(basename("a/b/c.go")) // "c"
fmt.Println(basename("c.d.go"))   // "c.d"
fmt.Println(basename("abc"))      // "abc"
```

Первая версия `basename` выполняет всю работу самостоятельно, без помощи библиотек:

gopl.io/ch3/basename1

```
// basename убирает компоненты каталога и суффикс типа файла.
// a => a, a.go => a, a/b/c.go => c, a/b.c.go => b.c

func basename(s string) string {
    // Отбрасываем последний символ '/' и все перед ним.
    for i := len(s) - 1; i >= 0; i-- {
        if s[i] == '/' {
            s = s[i+1:]
            break
        }
    }
    // Сохраняем все до последней точки '.'.
    for i := len(s) - 1; i >= 0; i-- {
        if s[i] == '.' {
            s = s[:i]
            break
        }
    }
    return s
}
```

Более простая версия использует библиотечную функцию `strings.LastIndex`:

gopl.io/ch3/basename2

```
func basename(s string) string {
    slash := strings.LastIndex(s, "/") // -1, если "/" не найден
    s = s[slash+1:]
    if dot := strings.LastIndex(s, "."); dot >= 0 {
        s = s[:dot]
    }
    return s
}
```

Пакеты `path` и `path/filepath` предоставляют более общий набор функций для работы с иерархическими именами. Пакет `path` работает с путями с косой чертой в качестве разделителя на любой платформе. Он не должен использоваться для имен файлов, но подходит для других вещей, таких как компоненты пути в URL. Пакет `path/filepath`, напротив, используя правила конкретной платформы, работает с именами файлов, такими как `/foo/bar` для POSIX или `c:\foo\bar` в Microsoft Windows.

Давайте рассмотрим еще один пример работы с подстроками. Задача заключается в том, чтобы взять строковое представление целого числа, такое как `"12345"`, и вставить запятые, разделяющие каждые три цифры, как в строке `"12,345"`[1]. Эта версия работает только с целыми числами; обработка чисел с плавающей точкой остается читателям в качестве упражнения.

gopl.io/ch3/comma

```
// comma вставляет запятые в строковое представление
// неотрицательного десятичного числа.

func comma(s string) string {
    n := len(s)
    if n <= 3 {
        return s
    }
    return comma(s[:n-3]) + "," + s[n-3:]
}
```

Аргументом функции `comma` является строка. Если ее длина меньше или равна 3, запятая не нужна. В противном случае функция `comma` рекурсивно вызывает саму себя с подстрокой, состоящей из всех символов, кроме трех последних, и прибавляет к результату рекурсивного вызова запятую и эти последние три символа.

Строка содержит массив байтов, который, будучи созданным, является неизменяемым. Элементы байтового среза, напротив, можно свободно модифицировать.

Строки можно преобразовывать в байтовые срезы и обратно:

```
s  := "abc"
b  := []byte(s)
s2 := string(b)
```

[1] Разделение групп разрядов, принятое в США. — *Примеч. пер.*

Концептуально преобразование `[]byte(s)` выделяет память для нового массива байтов, хранящего копию байтов строки `s`, и дает срез, ссылающийся на весь этот массив. Оптимизирующий компилятор в некоторых случаях может избежать выделения памяти и копирования, но в общем случае копирование необходимо для того, чтобы байты строки `s` оставались неизменными, даже если байты массива `b` будут модифицированы. Преобразование из байтового среза в строку с помощью `string(b)` также создает копию, чтобы гарантировать неизменяемость результирующей строки `s2`.

Чтобы избежать преобразований и излишних выделений памяти, многие вспомогательные функции в пакете `bytes` являются двойниками функций из пакета `strings`. Например, вот несколько функций из пакета `strings`:

```
func Contains(s, substr string) bool
func Count(s, sep string) int
func Fields(s string) []string
func HasPrefix(s, prefix string) bool
func Index(s, sep string) int
func Join(a []string, sep string) string
```

А вот соответствующие функции из пакета `bytes`:

```
func Contains(b, subslice []byte) bool
func Count(s, sep []byte) int
func Fields(s []byte) [][]byte
func HasPrefix(s, prefix []byte) bool
func Index(s, sep []byte) int
func Join(s [][]byte, sep []byte) []byte
```

Единственным их различием является то, что строки в них заменены срезами байтов.

Пакет `bytes` предоставляет тип `Buffer` для эффективной работы со срезами байтов. Изначально `Buffer` пуст, но растет по мере того, как в него записываются данные типов наподобие `string`, `byte` или `[]byte`. Как показано в примере ниже, переменная `bytes.Buffer` не требует инициализации, поскольку нулевое значение данного типа вполне используемо:

gopl.io/ch3/printints

```
// intsToString подобна fmt.Sprint(values), но добавляет запятые.
func intsToString(values []int) string {
    var buf bytes.Buffer
    buf.WriteByte('[')
    for i, v := range values {
        if i > 0 {
            buf.WriteString(", ")
        }
        fmt.Fprintf(&buf, "%d", v)
    }
    buf.WriteByte(']')
    return buf.String()
}
```

```
func main() {
    fmt.Println(intsToString([]int{1, 2, 3})) // "[1, 2, 3]"
}
```

Когда к `bytes.Buffer` добавляется произвольная руна в кодировке UTF-8, лучше использовать его метод `WriteRune`; метод `WriteByte` хорошо подходит для ASCII-символов, таких как `'['` и `']'`.

Тип `bytes.Buffer` чрезвычайно универсален, и когда в главе 7, "Интерфейсы", мы будем рассматривать интерфейсы, то увидим, как он может быть использован в качестве замены файла там, где требуются операции ввода-вывода для записи байтов (`io.Writer`), как это делает функция `Fprintf`, или для их чтения (`io.Reader`).

Упражнение 3.10. Напишите нерекурсивную версию функции `comma`, использующую `bytes.Buffer` вместо конкатенации строк.

Упражнение 3.11. Усовершенствуйте функцию `comma` так, чтобы она корректно работала с числами с плавающей точкой и необязательным знаком.

Упражнение 3.12. Напишите функцию, которая сообщает, являются ли две строки анаграммами одна другой, т.е. состоят ли они из одних и тех же букв в другом порядке.

3.5.5. Преобразования между строками и числами

В дополнение к преобразованиям между строками, рунами и байтами часто требуются преобразования между числовыми значениями и их строковыми представлениями. Это делается с помощью функций из пакета `strconv`.

Для преобразования целого числа в строку можно воспользоваться функцией `fmt.Sprintf`; другой вариант — функция `strconv.Itoa` ("integer to ASCII"):

```
x := 123
y := fmt.Sprintf("%d", x)
fmt.Println(y, strconv.Itoa(x)) // "123 123"
```

Для форматирования чисел в другой системе счисления можно использовать функции `FormatInt` и `FormatUint`:

```
fmt.Println(strconv.FormatInt(int64(x), 2)) // "1111011"
```

Символы преобразования `%b`, `%d`, `%o` и `%x` функции `fmt.Printf` часто более удобны, чем функции `Format`, в особенности если мы хотим добавить какую-то информацию, помимо самого числа:

```
s := fmt.Sprintf("x=%b", x) // "x=1111011"
```

Для анализа строкового представления целого числа используйте такие функции пакета `strconv`, как `Atoi` или `ParseInt`, а для беззнаковых чисел — `ParseUint`:

```
x, err := strconv.Atoi("123")             // x имеет тип int
// В десятичной системе счисления, до 64 битов:
y, err := strconv.ParseInt("123", 10, 64)
```

Третий аргумент функции `ParseInt` дает размер целочисленного типа, в который должен вписываться результат; так, 16 подразумевает тип `int16`, а особое значение 0

означает тип `int`. В любом случае типом результата `y` всегда является `int64`, который затем можно преобразовать в тип меньшего размера.

Иногда для анализа входной информации, состоящей из упорядоченной комбинации строк и чисел в одной строке может пригодиться функция `fmt.Scanf`, но это решение может быть негибким, особенно при обработке неполного или неправильного ввода.

3.6. Константы

Константы представляют собой выражения, значения которых известны компилятору и вычисление которых гарантированно происходит во время компиляции, а не во время выполнения. Базовым типом каждой константы является фундаментальный тип: логическое значение, строка или число.

Объявление `const` определяет именованные значения, которые синтаксически выглядят как переменные, значения которых являются константами, что предотвращает их случайное (или ошибочное) изменение во время выполнения программы. Например, для представления такой математической константы, как число π, константа предпочтительнее переменной, поскольку значение этого числа не изменяется:

```
const pi = 3.14159 // Лучшим приближением является math.Pi
```

Как и в случае переменных, в одном объявлении может находиться последовательность констант; такое объявление подходит для групп связанных значений:

```
const (
    e  = 2.71828182845904523536028747135266249775724709369995957496696
    pi = 3.14159265358979323846264338327950288419716939937510582097494
)
```

Многие вычисления констант могут быть полностью завершены во время компиляции, уменьшая тем самым количество работы, необходимой во время выполнения, и позволяя выполнять другие оптимизации компилятора. При этом ошибки, обычно обнаруживаемые во время выполнения, могут быть найдены во время компиляции, если операнды представляют собой константы, например целочисленное деление на нуль, индексация за пределами строки или любые операции с плавающей точкой, которые приводят к значению, не являющемуся корректным конечным значением с плавающей точкой.

Результаты всех арифметических и логических операций, а также операций сравнения с операндами-константами также являются константами, как и результаты преобразований и вызовов некоторых встроенных функций, таких как `len`, `cap`, `real`, `imag`, `complex` и `unsafe.Sizeof` (раздел 13.1).

Поскольку компилятору известны их значения, константные выражения могут находиться в типах, в частности — в качестве длины типа массива:

```
const IPv4Len = 4

// parseIPv4 анализирует адрес IPv4 (d.d.d.d).
```

```
func parseIPv4(s string) IP {
    var p [IPv4Len]byte
    // ...
}
```

Объявление константы может указывать как тип, так и значение, но при отсутствии явно указанного типа он выводится из выражения в правой части. В приведенном далее исходном тексте `time.Duration` представляет собой именованный тип, базовым для которого является тип `int64`, а `time.Minute` — константа этого типа. Таким образом, обе объявленные ниже константы имеют тип `time.Duration`, что показывают символы преобразования `%T`:

```
const noDelay time.Duration = 0
const timeout = 5 * time.Minute
fmt.Printf("%T %[1]v\n", noDelay)     // "time.Duration 0"
fmt.Printf("%T %[1]v\n", timeout)     // "time.Duration 5m0s"
fmt.Printf("%T %[1]v\n", time.Minute) // "time.Duration 1m0s"
```

Если последовательность констант объявлена в виде группы, выражение с правой стороны от символа = может быть опущено для всех констант, кроме первой. При этом подразумевается, что в текущей константе без явно указанного выражения будут использованы значение и тип предыдущего выражения, например:

```
const (
    a = 1
    b
    c = 2
    d
)
fmt.Println(a, b, c, d) // "1 1 2 2"
```

Если все значения констант оказываются одинаковыми, от этого не слишком много толку. Гораздо интереснее и полезнее использовать генератор констант `iota`.

3.6.1. Генератор констант `iota`

Объявление `const` может использовать *генератор констант* `iota`, который применяется для создания последовательности связанных значений без их явного указания. В объявлении `const` значение `iota` начинается с нуля и увеличивается на единицу для каждого элемента в последовательности.

Вот пример из пакета `time`, который определяет именованные константы типа `Weekday` для дней недели, начиная с нуля для `Sunday`. Типы такого рода часто именуются *перечислениями*.

```
type Weekday int
const (
    Sunday Weekday = iota
    Monday
    Tuesday
```

```
    Wednesday
    Thursday
    Friday
    Saturday
)
```

Это объявление определяет `Sunday` как 0, `Monday` — как 1 и т.д.

Можно использовать `iota` и в более сложных выражениях, как в приведенном далее примере из пакета `net`, где каждому из младших пяти битов целого беззнакового числа присваивается уникальное имя:

```
type Flags uint
const (
    FlagUp Flags = 1 << iota // is up
    FlagBroadcast
    FlagLoopback
    FlagPointToPoint
    FlagMulticast
)
```

С увеличением `iota` каждая константа получает значение `1 << iota` (т.е. получается последовательность возрастающих степеней двойки), каждое из которых соответствует одному отдельному биту. Эти константы можно использовать в функциях, которые выполняют установку, сброс или проверку одного или нескольких битовых флагов:

gopl.io/ch3/netflag

```
func IsUp(v Flags) bool     { return v&FlagUp == FlagUp }
func TurnDown(v *Flags)     { *v &^= FlagUp }
func SetBroadcast(v *Flags) { *v |= FlagBroadcast }
func IsCast(v Flags) bool   { return v&(FlagBroadcast|FlagMulticast)!=0 }
func main() {
    var v Flags = FlagMulticast | FlagUp
    fmt.Printf("%b %t\n", v, IsUp(v))   // "10001 true"
    TurnDown(&v)
    fmt.Printf("%b %t\n", v, IsUp(v))   // "10000 false"
    SetBroadcast(&v)
    fmt.Printf("%b %t\n", v, IsUp(v))   // "10010 false"
    fmt.Printf("%b %t\n", v, IsCast(v)) // "10010 true"
}
```

В качестве еще более сложного примера применения `iota` ниже показано объявление имен для степеней значения 1024:

```
const (
    _ = 1 << (10 * iota)
    KiB   // 1024
    MiB   // 1048576
```

```
    GiB   // 1073741824
    TiB   // 1099511627776 (превышает 1 << 32)
    PiB   // 1125899906842624
    EiB   // 1152921504606846976
    ZiB   // 1180591620717411303424 (превышает 1 << 64)
    YiB   // 1208925819614629174706176
)
```

Механизм `iota` имеет свои пределы. Например, невозможно генерировать более знакомые степени 1000 (KB, MB и т.д.), поскольку отсутствует оператор возведения в степень.

Упражнение 3.13. Напишите объявления `const` для KB, MB и так далее до YB настолько компактно, насколько сможете.

3.6.2. Нетипизированные константы

Константы в Go немного необычны. Хотя константа может иметь любой фундаментальный тип данных, такой как `int` или `float64`, включая именованные фундаментальные типы, такие как `time.Duration`, многие константы не привязаны к определенному типу. Компилятор представляет эти не привязанные к типу константы с гораздо большей числовой точностью, чем значения фундаментальных типов, а их арифметика является более точной, чем машинная; можно предположить по крайней мере 256-битовую точность. Имеется шесть вариантов таких несвязанных констант, именуемых *нетипизированным* булевым значением, нетипизированным целым числом, нетипизированной руной, нетипизированным числом с плавающей точкой, нетипизированным комплексным числом и нетипизированной строкой.

Откладывая привязку к типу, нетипизированные константы могут не только сохранить высокую точность значений до более позднего момента, но и участвовать в гораздо большем количестве выражений без необходимости преобразования, чем связанные константы. Например, значения `ZiB` и `YiB` в приведенном выше примере слишком велики, чтобы хранить их в любой переменной целочисленного типа, но они являются корректными константами, которые могут быть использованы в выражениях наподобие следующего:

```
fmt.Println(YiB/ZiB) // "1024"
```

В качестве еще одного примера константа с плавающей точкой `math.Pi` может использоваться везде, где требуется значение с плавающей точкой или комплексное значение:

```
var x float32    = math.Pi
var y float64    = math.Pi
var z complex128 = math.Pi
```

Если `math.Pi` будет связан с конкретным типом, таким как `float64`, результат будет не таким точным и будет требоваться преобразование типа при использовании там, где нужно значение типа `float32` или `complex128`:

```
const Pi64 float64 = math.Pi

var x float32    = float32(Pi64)
var y float64    = Pi64
var z complex128 = complex128(Pi64)
```

В случае литералов вариант нетипизированной константы определяется синтаксисом. Литералы `0`, `0.0`, `0i` и `'\u0000'` обозначают константы с одинаковыми значениями, но это разные варианты констант: нетипизированное целое значение, нетипизированное значение с плавающей точкой, нетипизированное комплексное значение и нетипизированная руна соответственно. Аналогично `true` и `false` представляют собой нетипизированные булевы значения, а строковые литералы являются нетипизированными строками.

Вспомните, что оператор `/` может представлять в зависимости от операндов как целочисленное деление, так и деление с плавающей точкой. Следовательно, выбор литерала может повлиять на результат выражения константного деления:

```
var f float64 = 212
fmt.Println((f-32)*5/9)     // "100"; (f-32)*5 имеет тип float64
fmt.Println(5/9*(f-32))     // "0"; 5/9 - нетипизированное целое (0)
fmt.Println(5.0/9.0*(f-32)) // "100"; 5.0/9.0 - нетипизированное
                            // значение с плавающей точкой
```

Нетипизированными могут быть только константы. Когда нетипизированная константа находится в правой части объявления переменной с явным указанием типа, как в первой инструкции приведенного ниже исходного текста, или присваивается переменной, как в следующих трех инструкциях, константа неявно преобразуется в тип этой переменной, если такое преобразование возможно.

```
var f float64 = 3 + 0i // Нетипизированное комплексное    -> float64
f = 2                  // Нетипизированное целое          -> float64
f = 1e123              // Нетипизированное действительное -> float64
f = 'a'                // Нетипизированная руна           -> float64
```

Таким образом, приведенные выше инструкции эквивалентны следующим:

```
var f float64 = float64(3 + 0i)
f = float64(2)
f = float64(1e123)
f = float64('a')
```

Явное или неявное, преобразование константы из одного типа в другой требует, чтобы целевой тип мог представлять исходное значение. Округление допускается для действительных и комплексных чисел с плавающей точкой.

```
const (
    deadbeef = 0xdeadbeef // Нетипизированный int = 3735928559
    a = uint32(deadbeef)  // uint32 = 3735928559
    b = float32(deadbeef) // float32 = 3735928576 (округление)
    c = float64(deadbeef) // float64 = 3735928559 (точное)
```

```
    d = int32(deadbeef)   // Ошибка: переполнение int32
    e = float64(1e309)    // Ошибка: переполнение float64
    f = uint(-1)          // Ошибка: не представимое uint
)
```

В объявлении переменной без явно указанного типа (включая краткие объявления переменных) версия нетипизированной константы неявно определяет тип переменной по умолчанию, как показано в следующих примерах:

```
i := 0      // Нетипизированное целое; неявное int(0)
r := '\000' // Нетипизированная руна; неявное rune('\000')
f := 0.0    // Нетипизированное действительное; неявное float64(0.0)
c := 0i     // Нетипизированное комплексное; неявное complex128(0i)
```

Обратите внимание на асимметрию: нетипизированные целые значения преобразуются в тип `int`, размер которого не гарантируется, но нетипизированные значения с плавающей точкой и нетипизированные комплексные значения преобразуются в типы с явно указанным размером `float64` и `complex128`. Язык не имеет "безразмерных" типов `float` и `complex`, аналогичных типу `int`, поскольку очень трудно писать корректные численные алгоритмы, не зная размер типов данных с плавающей точкой.

Чтобы придать переменной другой тип, необходимо явно преобразовать нетипизированную константу в требуемый тип или указать требуемый тип в объявлении переменной, как в приведенных примерах:

```
var i = int8(0)
var i int8 = 0
```

Эти типы по умолчанию особенно важны при преобразовании нетипизированной константы в значение интерфейса (см. главу 7, "Интерфейсы"), так как они определяют его динамический тип.

```
fmt.Printf("%T\n", 0)      // "int"
fmt.Printf("%T\n", 0.0)    // "float64"
fmt.Printf("%T\n", 0i)     // "complex128"
fmt.Printf("%T\n", '\000') // "int32" (руна)
```

Итак, мы рассмотрели фундаментальные типы данных Go. Следующий шаг заключается в их объединении в более крупные группы, такие как массивы и структуры, а затем в структуры данных для решения реальных задач. Это и есть тема главы 4, "Составные типы".

4

Составные типы

В главе 3, "Фундаментальные типы данных", мы обсудили фундаментальные типы, которые служат в качестве строительных блоков для структур данных в программах Go; они являются атомами нашей Вселенной. В этой главе мы рассмотрим составные типы, молекулы, созданные путем объединения фундаментальных типов различными способами. Мы будем говорить о четырех таких типах — массивах, срезах, отображениях и структурах, а в конце этой главы покажем, как с их помощью кодировать и анализировать структурированные данные, получаемые из данных JSON, и генерировать HTML-текст из шаблонов.

Массивы и структуры являются *составными* (*агрегированными*) типами данных; их значения создаются путем конкатенации в памяти других значений. Массивы гомогенны (все элементы массива имеют один и тот же тип), в то время как структуры гетерогенны. И массивы, и структуры имеют фиксированный размер. В отличие от них, срезы и отображения являются динамическими структурами данных, растущими по мере добавления в них значений.

4.1. Массивы

Массив представляет собой последовательность фиксированной длины из нуля или более элементов определенного типа. Из-за фиксированной длины массивы редко используются в Go непосредственно. Срезы, которые могут увеличиваться и уменьшаться, являются гораздо более гибкими, но, чтобы понять срезы, сначала следует разобраться в массивах.

Доступ к отдельным элементам массива осуществляется с помощью обычных обозначений индексирования, значения индексов в которых имеют значения от нуля до значения, на единицу меньшего длины массива. Встроенная функция `len` возвращает количество элементов в массиве.

```
var a [3]int             // Массив из трех целых чисел
fmt.Println(a[0])        // Вывод первого элемента
fmt.Println(a[len(a)-1]) // Вывод последнего элемента, a[2]

// Вывод значений индексов и элементов.
for i, v := range a {
```

```
    fmt.Printf("%d %d\n", i, v)
}
// Вывод только элементов.
for _, v := range a {
    fmt.Printf("%d\n", v)
}
```

По умолчанию элементам новой переменной массива изначально присваиваются нулевые значения типа элемента (для чисел это значение `0`). Для инициализации массива списком значений можно использовать *литерал массива*:

```
var q [3]int = [3]int{1, 2, 3}
var r [3]int = [3]int{1, 2}
fmt.Println(r[2])     // "0"
```

Если в литерале массива на месте длины находится троеточие "`...`", то длина массива определяется количеством инициализаторов. Определение `q` можно упростить до

```
q := [...]int{1, 2, 3}
fmt.Printf("%T\n", q) // "[3]int"
```

Размер массива является частью его типа, так что типы `[3]int` и `[4]int` различны. Размер должен быть константным выражением, т.е. выражением, значение которого может быть вычислено во время компиляции программы.

```
q := [3]int{1, 2, 3}
q = [4]int{1, 2, 3, 4} // Ошибка компиляции: нельзя присвоить
                       // [4]int переменной типа [3]int
```

Как мы увидим, синтаксис литерала для массивов, срезов, отображений и структур подобен. Выше используется упорядоченный список значений, но можно также указать список пар "индекс–значение":

```
type Currency int

const (
    USD Currency = iota
    EUR
    GBP
    RUR
)

symbol := [...]string{USD: "$", EUR: "€", GBP: "£", RUR: "₽"}
fmt.Println(RUR, symbol[RUR]) // "3 ₽"
```

В этом случае индексы могут появляться в любом порядке, а некоторые из них могут быть опущены; как и прежде, неуказанные значения получают нулевое значение типа элемента. Например,

```
r := [...]int{99:-1}
```

определяет массив r со ста элементами, среди которых ненулевым является только последний элемент, значение которого равно –1.

Если тип элемента массива является *сравниваемым*, то таким же является и тип массива, так что мы можем сравнить два массива такого типа непосредственно, с помощью оператора ==, который сообщает, все ли соответствующие элементы массивов равны. Оператор != является отрицанием оператора ==.

```
a := [2]int{1, 2}
b := [...]int{1, 2}
c := [2]int{1, 3}
fmt.Println(a == b, a == c, b == c) // "true false false"
d := [3]int{1, 2}
fmt.Println(a == d) // Ошибка компиляции: разные типы [2]int и [3]int
```

В качестве более правдоподобного примера функция Sum256 из пакета crypto/sha256 генерирует криптографический хеш, или *дайджест*, SHA256 сообщения, хранящегося в произвольном байтовом срезе. Дайджест состоит из 256 битов, поэтому его типом является [32]byte. Если два дайджеста совпадают, то очень вероятно, что соответствующие сообщения одинаковы; если же дайджесты различаются, то различаются и сообщения. Приведенная далее программа выводит и сравнивает дайджесты SHA256 для "x" и "X":

gopl.io/ch4/sha256

```
import "crypto/sha256"

func main() {
    c1 := sha256.Sum256([]byte("x"))
    c2 := sha256.Sum256([]byte("X"))
    fmt.Printf("%x\n%x\n%t\n%T\n", c1, c2, c1 == c2, c1)

    // Вывод:
    // 2d711642b726b04401627ca9fbac32f5c8530fb1903cc4db02258717921a4881
    // 4b68ab3847feda7d6c62c1fbcbeebfa35eab7351ed5e78f4ddadea5df64b8015
    // false
    // [32]uint8
}
```

Эти входные данные различаются только одним битом, но в дайджестах различны примерно половина битов. Обратите внимание на символы преобразования в Printf: %x выводит все элементы массива или среза байтов в шестнадцатеричном формате, %t выводит булево значение, а %T показывает тип значения.

При вызове функции копия каждого значения аргумента присваивается переменной соответствующего параметра, так что функция получает копию, а не оригинал. Передача таким образом больших массивов может быть неэффективной, а любые изменения, вносимые функцией в элементы массива, будут влиять только на копию, но не на оригинал. Поэтому Go работает с массивами как с любым другим типом, но это поведение отличается от поведения языков, которые неявно передают массивы *по ссылке*.

Конечно, можно явно передать указатель на массив, так что любые изменения, которые функция будет делать в элементах массива, будут видны вызывающей функции. Приведенная ниже функция заполняет нулями содержимое массива `[32]byte`:

```
func zero(ptr *[32]byte) {
    for i := range ptr {
        ptr[i] = 0
    }
}
```

Литерал массива `[32]byte{}` дает нам массив из 32 байтов. Каждый элемент массива имеет нулевое значение для типа `byte`, которое просто равно нулю. Мы можем использовать этот факт для написания другой версии функции `zero`:

```
func zero(ptr *[32]byte) {
    *ptr = [32]byte{}
}
```

Применение указателя на массив оказывается эффективным и позволяет вызываемой функции изменять переменные вызывающей функции, но массивы остаются негибким решением из-за присущего им фиксированного размера. Например, функция `zero` не примет указатель на переменную `[16]byte`; нет также никакого способа добавления или удаления элементов массива. По этим причинам, за исключением особых случаев, таких как хеш `SHA256` фиксированного размера, массивы в качестве параметров функции используются редко; вместо этого обычно используются срезы.

Упражнение 4.1. Напишите функцию, которая подсчитывает количество битов, различных в двух дайджестах SHA256 (см. `PopCount` в разделе 2.6.2).

Упражнение 4.2. Напишите программу, которая по умолчанию выводит дайджест SHA256 для входных данных, но при использовании соответствующих флагов командной строки выводит SHA384 или SHA512.

4.2. Срезы

Срезы представляют собой последовательности переменной длины, все элементы которых имеют один и тот же тип. Тип среза записывается как `[]T`, где `T` — тип элементов среза; он выглядит как тип массива без указания размера.

Массивы и срезы тесно связаны. Срез — это легковесная структура данных, которая предоставляет доступ к подпоследовательности элементов массива (или, возможно, ко всем элементам), известного как *базовый массив*. Срез состоит из трех компонентов: указателя, длины и емкости. Указатель указывает на первый элемент массива, доступный через срез (который не обязательно совпадает с первым элементом массива). Длина — это количество элементов среза; она не может превышать емкость, которая, как правило, представляет собой количество элементов между началом среза и концом базового массива. Эти значения возвращаются встроенными функциями `len` и `cap`.

Несколько срезов могут совместно использовать один и тот же базовый массив и относиться к перекрывающимся частям этого массива. На рис. 4.1 показан массив строк для месяцев года и два перекрывающихся среза. Массив объявляется как

```
months := [...]string{1: "Январь", /* ... */, 12: "Декабрь"}
```

так что январь соответствует элементу `months[1]`, а декабрь — `months[12]`. Как обычно, первое значение находится в элементе массива с индексом 0, но поскольку месяцы всегда нумеруются начиная с 1, инициализацию нулевого элемента в объявлении можно опустить, и он будет инициализирован пустой строкой.

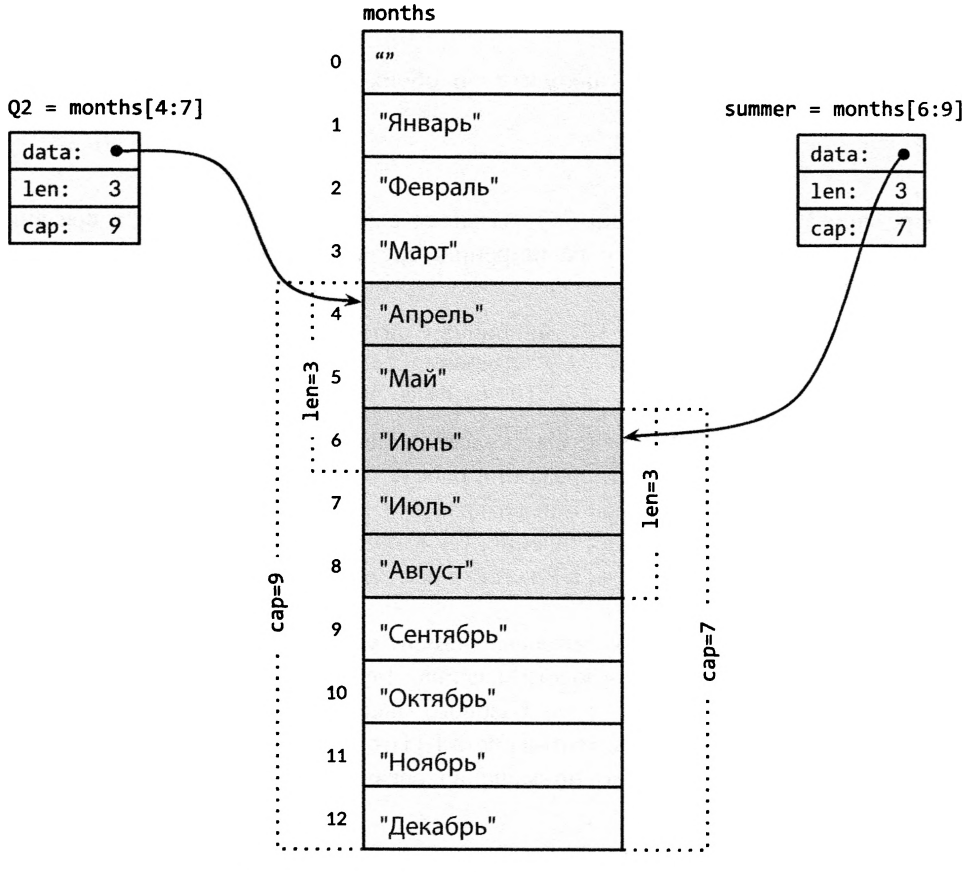


Рис. 4.1. Два перекрывающихся среза массива месяцев

Оператор среза `s[i:j]`, где `0 ≤ i ≤ j ≤ cap(s)`, создает новый срез, ссылающийся на элементы последовательности `s` с `i` по `j-1`. Последовательность может быть переменной массива, указателем на массив или другим срезом. Получаемый в результате срез имеет `j-i` элементов. Если значение `i` опущено, используется значение 0; если опущено значение `j`, используется значение `len(s)`. Таким образом, срез `months[1:13]` содержит весь диапазон допустимых месяцев, как и срез

months[1:]; срез months[:] указывает на весь массив. Давайте определим перекрывающиеся срезы для второго квартала и для лета в северном полушарии:

```
Q2 := months[4:7]
summer := months[6:9]
fmt.Println(Q2)     // ["Апрель" "Май" "Июнь"]
fmt.Println(summer) // ["Июнь" "Июль" "Август"]
```

Июнь включен в оба среза и является единственным выводом приведенного далее (неэффективного) поиска общих элементов:

```
for _, s := range summer {
    for _, q := range Q2 {
        if s == q {
            fmt.Printf("%s находится в обоих срезах\n", s)
        }
    }
}
```

"Срезание" за пределами cap(s) вызывает аварийную ситуацию, а срезание за пределами len(s) приводит к расширению среза, так что результат может быть длиннее оригинала:

```
fmt.Println(summer[:20])    // Аварийная ситуация: выход за диапазон
endlessSummer := summer[:5] // Расширение среза (в пределах диапазона)
fmt.Println(endlessSummer)  // "[Июнь Июль Август Сентябрь Октябрь]"
```

В качестве отступления отметим сходство операции получения подстроки при работе со строками и операции среза при работе со срезами []byte. Обе операции записываются как x[m:n] и обе они возвращают подпоследовательности исходных байтов, используя базовое представление, так что обе операции выполняются за константное время. Выражение x[m:n] дает строку, если x — это строка, и []byte, если x — это объект []byte.

Поскольку срез содержит указатель на элемент массива, передача среза в функцию позволяет изменять элементы базового массива. Другими словами, копирование среза создает *псевдоним* (раздел 2.3.2) для базового массива. Приведенная ниже функция reverse обращает порядок элементов среза []int на месте, без привлечения дополнительной памяти, и может быть применена к срезам любой длины.

gopl.io/ch4/rev

```
// reverse обращает порядок чисел "на месте"
func reverse(s []int) {
    for i, j := 0, len(s) - 1; i < j; i, j = i+1, j-1
    {
        s[i], s[j] = s[j], s[i]
    }
}
```

Вот как выглядит обращение порядка элементов массива a:

```
a := [...]int{0, 1, 2, 3, 4, 5}
reverse(a[:])
fmt.Println(a) // "[5 4 3 2 1 0]"
```

Чтобы выполнить *циклический сдвиг* среза влево на *n* элементов, можно просто трижды применить функцию `reverse`: сначала — к первым *n* элементам, затем — к оставшимся и наконец — ко всему срезу. (Для циклического сдвига вправо первым выполняется третий вызов.)

```
s := []int{0, 1, 2, 3, 4, 5}
// Циклический сдвиг влево на две позиции.
reverse(s[:2])
reverse(s[2:])
reverse(s)
fmt.Println(s) // "[2 3 4 5 0 1]"
```

Обратите внимание, как выражение, которое инициализирует срез `s`, отличается от такового для массива `a`. *Литерал среза* выглядит как литерал массива, в виде последовательности значений, разделенных запятыми и заключенных в фигурные скобки, но размер при этом не указан. Тем самым неявно создаются переменная массива правильного размера, и срез, который на него указывает. Как и в случае литералов массива, литералы срезов могут указывать значения упорядоченно или явно указывать их индексы, или использовать сочетание этих двух стилей.

В отличие от массивов срезы не являются сравниваемыми, поэтому мы не можем использовать оператор ==, чтобы проверить, содержат ли два среза одинаковые элементы. Для сравнения двух срезов байтов (`[]byte`) стандартная библиотека предоставляет высокооптимизированную функцию `bytes.Equal`, но для других типов срезов необходимо выполнять сравнение самостоятельно:

```
func equal(x, y []string) bool {
    if len(x) != len(y) {
        return false
    }
    for i := range x {
        if x[i] != y[i] {
            return false
        }
    }
    return true
}
```

Глядя на естественность этой "глубокой" проверки равенства и на то, что стоимость ее выполнения не превышает таковую для оператора == для массивов строк, может показаться странным, почему точно так же не выполняется и сравнение срезов. Есть две причины, по которым проверка глубокой эквивалентности является проблематичной. Во-первых, в отличие от элементов массива элементы среза являются косвенными, что позволяет срезу содержать самого себя. Хотя есть способы обработ-

ки таких ситуаций, ни один из них не является простым, эффективным и, что самое важное, очевидным.

Во-вторых, в силу косвенности элементов фиксированное значение среза может содержать различные элементы в разные моменты времени при изменении содержимого базового массива. Поскольку хеш-таблицы, такие как тип отображения Go, делают только поверхностные копии своих ключей, требуется, чтобы равенство для каждого ключа оставалось неизменным на протяжении всего времени жизни хеш-таблицы. Таким образом, глубокая эквивалентность делает срезы неподходящими для использования в качестве ключей отображений. Для ссылочных типов, таких как указатели и каналы, операторы == проверяют *ссылочную тождественность*, т.е. ссылаются ли две сущности на один и тот же объект. Аналогично проверка "поверхностного" равенства для срезов может оказаться полезной и позволит решить проблему с отображениями, но такая несогласованная трактовка срезов и массивов оператором == будет запутывать программистов. Поэтому самым безопасным решением является полный запрет сравнения срезов.

Единственным разрешенным сравнением среза является сравнение с `nil`:

```
if summer == nil { /* ... */ }
```

Нулевым значением типа среза является значение `nil`. Такой срез не имеет базового массива. И длина, и емкость нулевого среза равны нулю, но нулевыми длиной и емкостью могут обладать и ненулевые срезы, такие как `[]int{}` или `make([]int,3)[3:]`. Как и в случае любого типа, который может иметь значение `nil`, это значение для конкретного среза можно записать с помощью выражения преобразования, такого как `[]int(nil)`:

```
var s []int     // len(s) == 0, s == nil
s = nil         // len(s) == 0, s == nil
s = []int(nil)  // len(s) == 0, s == nil
s = []int{}     // len(s) == 0, s != nil
```

Так что, если вам нужно проверить, является ли срез пустым, используйте проверку `len(s) == 0`, но не `s == nil`. За исключением сравнения со значением `nil` нулевой срез ведет себя как любой другой срез нулевой длины; например, вызов `reverse(nil)` является совершенно безопасным. Если не задокументировано обратное, функции Go должны рассматривать все срезы нулевой длины одинаково, независимо от того, равны они `nil` или нет.

Встроенная функция `make` создает срез с определенным типом элементов, длиной и емкостью. Аргумент, указывающий емкость, может быть опущен, и в таком случае емкость принимается равной длине:

```
make([]T, len)
make([]T, len, cap) // то же, что и make([]T, cap)[:len]
```

За кулисами функция `make` создает неименованную переменную массива и возвращает его срез; сам массив доступен только через возвращаемый срез. В первом случае срез является представлением всего массива целиком. Во втором срез является

представлением только первых len его элементов, но его емкость включает в себя весь массив. Дополнительные элементы отводятся для будущего роста среза.

4.2.1. Функция append

Встроенная функция append добавляет элементы в срез:

```
var runes []rune
for _, r := range "Hello, 世界" {
    runes = append(runes, r)
}
fmt.Printf("%q\n", runes) // "['H' 'e' 'l' 'l' 'o' ',' ' ' '世' '界']"
```

Цикл использует функцию append для построения среза из десяти рун, закодированных строковым литералом, хотя данная конкретная задача решается куда проще с помощью встроенного преобразования []rune("Hello, 世界").

Функция append имеет решающее значение для понимания работы срезов, так что давайте взглянем на то, что происходит в программе. Вот версия функции под названием appendInt, которая специализирована для срезов []int:

gopl.io/ch4/append

```
func appendInt(x []int, y int) []int {
    var z []int
    zlen := len(x) + 1
    if zlen <= cap(x) {
        // Имеется место для роста. Расширяем срез.
        z = x[:zlen]
    } else {
        // Места для роста нет. Выделяем новый массив. Увеличиваем
        // в два раза для линейной амортизированнной сложности.
        zcap := zlen
        if zcap < 2*len(x) {
            zcap = 2 * len(x)
        }
        z = make([]int, zlen, zcap)
        copy(z, x) // Встроенная функция; см. текст раздела
    }
    z[len(x)] = y
    return z
}
```

Каждый вызов appendInt должен проверять, имеет ли разрез достаточную емкость для добавления новых элементов в существующий массив. Если да, функция расширяет срез, определяя срез большего размера (но все еще в пределах исходного массива), копирует элемент y во вновь добавленную к срезу память и возвращает срез. Входной срез x и результирующий срез z при этом используют один и тот же базовый массив.

Если же для роста недостаточно памяти, функции `appendInt` необходимо выделить новый массив, достаточно большой, чтобы хранить результат, затем скопировать в него значения из `x` и добавить новый элемент `y`. Результат `z` теперь относится к другому базовому массиву, отличному от базового массива `x`.

Можно было бы просто скопировать элементы с помощью явного массива, но проще использовать встроенную функцию `copy`, которая копирует элементы из одного среза в другой того же типа. Ее первый аргумент — целевой срез, а второй — исходный, что напоминает порядок операндов в присваивании `dst = src`. Срезы могут относиться к одному и тому же базовому массиву; они могут даже перекрываться. Хотя мы и не используем его здесь, функция `copy` возвращает количество фактически скопированных элементов, которое представляет собой меньшую из длин двух срезов, поэтому опасность выйти за пределы диапазона отсутствует.

Для повышения эффективности новый массив обычно несколько больше, чем минимально необходимо для хранения `x` и `y`. Увеличение массива путем удвоения его размера при каждом расширении предотвращает чрезмерное количество выделений памяти и гарантирует, что добавление одного элемента в среднем выполняется за константное время. Приведенная далее программа демонстрирует это:

```
func main() {
    var x, y []int
    for i := 0; i < 10; i++ {
        y = appendInt(x, i)
        fmt.Printf("%d cap=%d\t%v\n", i, cap(y), y)
        x = y
    }
}
```

При каждом добавлении выводится выделенное количество памяти для массива:

```
0  cap=1   [0]
1  cap=2   [0 1]
2  cap=4   [0 1 2]
3  cap=4   [0 1 2 3]
4  cap=8   [0 1 2 3 4]
5  cap=8   [0 1 2 3 4 5]
6  cap=8   [0 1 2 3 4 5 6]
7  cap=8   [0 1 2 3 4 5 6 7]
8  cap=16  [0 1 2 3 4 5 6 7 8]
9  cap=16  [0 1 2 3 4 5 6 7 8 9]
```

Давайте подробнее рассмотрим итерацию `i=3`. Срез `x` содержит три элемента `[0 1 2]`, но имеет емкость `4`, поэтому имеется еще один незанятый элемент в конце массива, и функция `appendInt` может добавить элемент `3` без выделения памяти. Результирующий срез `y` имеет длину и емкость `4` и тот же базовый массив, что и у исходного среза `x`, как показано на рис. 4.2.

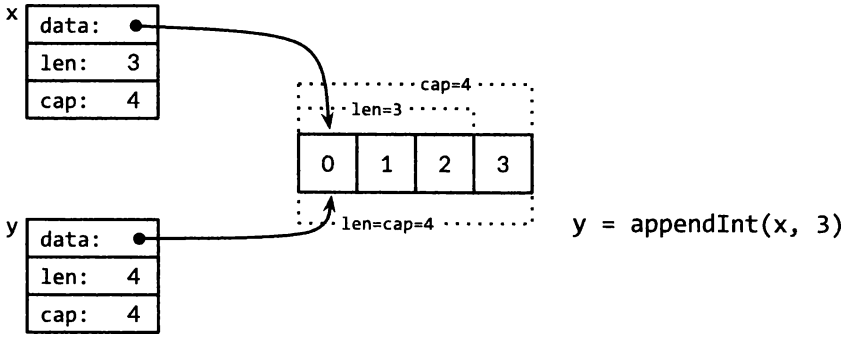


Рис. 4.2. Добавление при наличии места для роста

На следующей итерации, `i=4`, запаса пустых мест нет, так что функция `appendInt` выделяет новый массив размером 8, копирует в него четыре элемента `[0 1 2 3]` из `x` и добавляет значение `i`, равное 4. Результирующий срез `y` имеет длину 5 и емкость 8; остаток в 3 элемента предназначен для сохранения вставляемых значений на следующих трех итерациях без необходимости выделять для них память. Срезы `y` и `x` являются представлениями различных массивов. Эта операция изображена на рис. 4.3.

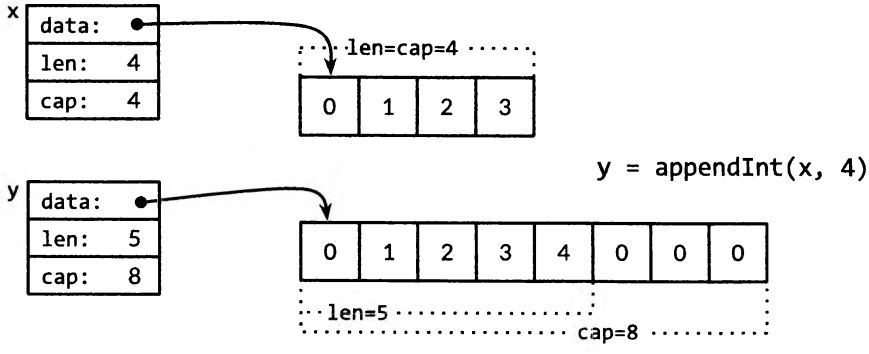


Рис. 4.3. Добавление при отсутствии места для роста

Встроенная функция `append` может использовать более сложные стратегии роста, чем функция `appendInt`. Обычно мы не знаем, приведет ли данный вызов функции `append` к перераспределению памяти, поэтому мы не можем считать ни что исходный срез относится к тому же массиву, что и результирующий срез, ни что он относится к другому массиву. Аналогично мы не должны предполагать, что операции над элементами старого среза будут (или не будут) отражены в новом срезе. Поэтому обычно выполняется присваивание результата вызова функции `append` той же переменной среза, которая была передана в функцию `append`:

```
runes = append(runes, r)
```

Обновление переменной среза требуется не только при вызове `append`, но и для любой функции, которая может изменить длину или емкость среза или сделать его

ссылающимся на другой базовый массив. Чтобы правильно использовать срезы, важно иметь в виду, что хотя элементы базового массива и доступны косвенно, указатель среза, его длина и емкость таковыми не являются. Чтобы обновить их, требуется присваивание, такое, как показано выше. В этом смысле срезы не являются "чисто" ссылочными типами, а напоминают составной тип, такой, как приведенная структура:

```
type IntSlice struct {
    ptr *int
    len, cap int
}
```

Наша функция `appendInt` добавляет в срез единственный элемент, но встроенная функция `append` позволяет добавлять больше одного нового элемента или даже целый их срез:

```
var x []int
x = append(x, 1)
x = append(x, 2, 3)
x = append(x, 4, 5, 6)
x = append(x, x...)      // Добавление среза x
fmt.Println(x)           // "[1 2 3 4 5 6 1 2 3 4 5 6]"
```

С помощью небольшой модификации, показанной ниже, можно обеспечить поведение нашей функции, совпадающее с поведением встроенной функции `append`. Троеточие `"..."` в объявлении функции `appendInt` делает ее *вариадической* функцией, т.е. функцией с переменным числом аргументов, которая может принимать любое количество завершающих аргументов. Такое троеточие в вызове `append` в фрагменте исходного текста выше показывает, как передавать список аргументов из среза. Мы подробно рассмотрим этот механизм в разделе 5.7.

```
func appendInt(x []int, y ...int) []int {
    var z []int
    zlen := len(x) + len(y)
    // ... расширяем z до как минимум длины zlen...
    copy(z[len(x):], y)
    return z
}
```

Логика расширения базового массива `z` остается неизменной и здесь не показана.

4.2.2. Работа со срезами "на месте"

Давайте рассмотрим больше примеров функций, которые, подобно функциям `rotate` и `reverse`, изменяют элементы среза "на месте", без привлечения дополнительной памяти.

Функция `nonempty` для данного списка строк возвращает только непустые строки:

```
gopl.io/ch4/nonempty
// Пример работы со срезом "на месте".
```

```
package main

import "fmt"

// nonempty возвращает срез, содержащий только непустые строки.
// Содержимое базового массива при работе функции изменяется.
func nonempty(strings []string) []string {
    i := 0
    for _, s := range strings {
        if s != "" {
            strings[i] = s
            i++
        }
    }
    return strings[:i]
}
```

Тонкостью данной функции является то, что входной и выходной срезы разделяют один и тот же базовый массив. Это позволяет избежать необходимости выделять еще один массив, хотя, конечно же, содержимое `data` частично перезаписывается, как доказывает вторая инструкция вывода:

```
data := []string{"one", "", "three"}
fmt.Printf("%q\n", nonempty(data)) // `["one" "three"]`
fmt.Printf("%q\n", data)           // `["one" "three" "three"]`
```

Таким образом, мы должны, как обычно, выполнять присваивание `data = nonempty(data)`.

Функцию `nonempty` можно написать с использованием функции `append`:

```
func nonempty2(strings []string) []string {
    out := strings[:0]   // Срез нулевой длины из исходного среза
    for _, s := range strings {
        if s != "" {
            out = append(out, s)
        }
    }
    return out
}
```

Какой бы вариант мы ни выбрали, такое использование массива требует, чтобы для каждого входного значения генерировалось не более одного выходного значения. Это требование выполняется для множества алгоритмов, которые отфильтровывают элементы последовательности или объединяют соседние. Такое сложное использование срезов является исключением, а не правилом, но в ряде случаев оно может оказаться ясным, эффективным и полезным.

Срез может быть использован для реализации стека. Для заданного изначально пустого среза `stack` поместить новое значение в стек можно с помощью функции `append`:

```
stack = append(stack, v)     // Внесение v в стек
```

Вершина стека представляет собой последний его элемент:

```
top := stack[len(stack) - 1] // Вершина стека
```

Снятие элемента со стека выглядит следующим образом:

```
stack = stack[:len(stack) - 1] // Удаление элемента из стека
```

Чтобы удалить элемент из средины среза, сохранив порядок оставшихся элементов, используйте функцию `copy` для переноса "вниз" на одну позицию элементов с более высокими номерами:

```
func remove(slice []int, i int) []int {
    copy(slice[i:], slice[i+1:])
    return slice[:len(slice)1]
}

func main() {
    s := []int{5, 6, 7, 8, 9}
    fmt.Println(remove(s, 2)) // "[5 6 8 9]"
}
```

Если нам не обязательно сохранять порядок, можно просто перенести последний элемент на место удаляемого:

```
func remove(slice []int, i int) []int {
    slice[i] = slice[len(slice)1]
    return slice[:len(slice)1]
}

func main() {
    s := []int{5, 6, 7, 8, 9}
    fmt.Println(remove(s, 2)) // "[5 6 9 8]
}
```

Упражнение 4.3. Перепишите функцию `reverse` так, чтобы вместо среза она использовала указатель на массив.

Упражнение 4.4. Напишите версию функции `rotate`, которая работает в один проход.

Упражнение 4.5. Напишите функцию, которая без выделения дополнительной памяти удаляет все смежные дубликаты в срезе `[]string`.

Упражнение 4.6. Напишите функцию, которая без выделения дополнительной памяти преобразует последовательности смежных пробельных символов Unicode (см. `unicode.IsSpace`) в срезе `[]byte` в кодировке UTF-8 в один пробел ASCII.

Упражнение 4.7. Перепишите функцию `reverse` так, чтобы она без выделения дополнительной памяти обращала последовательность символов среза `[]byte`, который представляет строку в кодировке UTF-8. Сможете ли вы обойтись без выделения новой памяти?

4.3. Отображения

Хеш-таблица является одной из самых гениальных и универсальных из всех структур данных. Это неупорядоченная коллекция пар "ключ–значение", в которой все ключи различны, а значение, связанное с заданным ключом, можно получить, обновить или удалить с использованием в среднем константного количества сравнений ключей, независимо от размера хеш-таблицы.

В Go *отображение* представляет собой ссылку на хеш-таблицу, а тип отображения записывается как `map[K]V`, где `K` и `V` являются типами его ключей и значений. Все ключи в данном отображении имеют один и тот же тип, как и все значения имеют один и тот же тип, но тип ключей не обязан совпадать с типом значений. Тип ключа `K` должен быть сравниваемым с помощью оператора `==`, чтобы отображение могло проверить, равен ли данный ключ одному из имеющихся в нем. Хотя числа с плавающей точкой являются сравниваемыми, сравнивать их на равенство — плохая идея, и, как мы упоминали в главе 3, "Фундаментальные типы данных", это особенно плохая идея, если NaN является возможным значением. На тип значения `V` никаких ограничений нет.

Встроенная функция `make` может использоваться для создания отображения:

```
ages := make(map[string]int) // Отображение строк на int
```

Для создания нового отображения, заполненного данными, можно использовать *литерал отображения*:

```
ages := map[string]int{
    "alice": 31,
    "charlie": 34,
}
```

Эта запись эквивалентна следующей:

```
ages := make(map[string]int)
ages["alice"] = 31
ages["charlie"] = 34
```

так что альтернативное выражение для создания нового пустого отображения имеет вид `map[string]int{}`.

Обратиться к элементам отображения можно с помощью обычной индексации:

```
ages["alice"] = 32
fmt.Println(ages["alice"]) // "32"
```

Удаление осуществляется с помощью встроенной функции `delete`:

```
delete(ages, "alice") // Удаление элемента ages["alice"]
```

Все эти операции безопасны, даже если элемент в отображении отсутствует; при использовании ключа, которого нет в отображении, поиск возвращает нулевое значение соответствующего типа. Так что, например, следующая инструкция работает,

даже когда ключа "bob" еще нет в отображении, поскольку значение ages["bob"] в этом случае будет равно 0:

```
ages["bob"] = ages["bob"] + 1 // С днем рождения!
```

Сокращенные записи x += y и x++ также работают с элементами отображения, поэтому приведенное выше выражение можно переписать как

```
ages["bob"] += 1
```

или даже как

```
ages["bob"]++
```

Однако элементы отображения не являются переменными, и мы не можем получить их адреса:

```
_ = &ages["bob"] // Ошибка: невозможно получить адрес в отображении
```

Одна из причин, по которым мы не можем получить адрес элемента отображения, заключается в том, что с ростом отображения может быть выполнено повторное хеширование элементов в новые места хранения, что потенциально делает адреса недействительными.

Для перечисления всех пар "ключ–значение" в отображении мы используем циклы по диапазону, аналогичные тем, которые мы видели для срезов. Последовательные итерации приведенного ниже цикла присваивают переменным name и age значения из очередной пары "ключ–значение":

```
for name, age := range ages {
    fmt.Printf("%s\t%d\n", name, age)
}
```

Порядок итераций по отображению не определен; различные реализации могут использовать разные хеш-функции, что приведет к иному порядку. На практике получается случайный порядок, который варьируется от одного запуска программы к другому. Это сделано преднамеренно; варьируемые последовательности помогают писать программы, которые одинаково надежны в разных реализациях языка. Для перечисления пар "ключ–значение" в определенном порядке требуется явная сортировка ключей, например, с помощью функции Strings из пакета sort, если ключи являются строками. Вот как выглядит общий шаблон:

```
import "sort"

var names []string
for name := range ages {
    names = append(names, name)
}
sort.Strings(names)
for _, name := range names {
    fmt.Printf("%s\t%d\n", name, ages[name])
}
```

Поскольку мы знаем окончательный размер names с самого начала, более эффективным решением является изначальное выделение массива требуемого размера. Приведенная далее инструкция создает срез, который изначально пуст, но обладает достаточной емкостью для хранения всех ключей из отображения ages:

```
names := make([]string, 0, len(ages))
```

В первом цикле по диапазону, показанном в исходном тексте выше, нам требуются только ключи отображения ages, поэтому мы опускаем вторую переменную цикла. Во втором цикле нам требуются только элементы среза names, поэтому мы используем пустой идентификатор _, чтобы игнорировать первую переменную (значение индекса).

Нулевым значением для типа отображения является nil:

```
var ages map[string]int
fmt.Println(ages == nil)    // "true"
fmt.Println(len(ages) == 0) // "true"
```

Большинство операций над отображениями, включая поиск, delete, len и цикл по диапазону, безопасно выполняются с нулевым отображением, поскольку оно ведет себя так же, как пустое отображение. Однако сохранение значений в нулевом отображении приводит к аварийной ситуации:

```
ages["carol"] = 21 // Аварийная ситуация: присваивание
                   // элементу нулевого отображения
```

Перед тем как выполнять присваивание, следует выделить память для отображения.

Доступ к элементу отображения с помощью индексации всегда дает значение. Если ключ присутствует в отображении, вы получаете соответствующее значение; если нет — вы получите нулевое значение типа элемента, как мы видели для ages["bob"]. Для многих целей это вполне нормально, но иногда нужно знать, есть ли некоторый элемент в отображении. Например, если типом элемента является число, то отличить несуществующий элемент от элемента, который имеет нулевое значение, можно с помощью следующего теста:

```
age, ok := ages["bob"]
if !ok {/*"bob" не является ключом в данном отображении; age == 0.*/}
```

Зачастую две такие инструкции объединяются, например, следующим образом:

```
if age, ok := ages["bob"]; !ok { /* ... */ }
```

Индексация отображения в этом контексте дает два значения. Второе значение — это логическое значение, показывающее, имеется ли данный элемент в отображении. Этой логической переменной часто дают имя ok, особенно если она сразу же используется в условии инструкции if.

Как и срезы, отображения нельзя сравнивать одно с другим; единственным законным сравнением является сравнение со значением nil. Чтобы проверить, содержат

ли два отображения одни и те же ключи и связанные с ними значения, мы должны написать цикл:

```
func equal(x, y map[string]int) bool {
    if len(x) != len(y) {
        return false
    }
    for k, xv := range x {
        if yv, ok := y[k]; !ok || yv != xv {
            return false
        }
    }
    return true
}
```

Обратите внимание, как мы используем `!ok` для того, чтобы отличить случаи "отсутствует" и "присутствует, но равен нулю". Если бы мы наивно написали `xv != y[k]`, то показанный ниже вызов сообщил бы, что аргументы равны, хотя это и не так:

```
// Истинно при некорректном написании функции.
equal(map[string]int{"A": 0}, map[string]int{"B": 42})
```

Go не предоставляет тип `set`, но поскольку все ключи отображения различны, отображение может служить и для этой цели. Для иллюстрации программа `dedup` считывает последовательность строк и выводит только первое вхождение каждой из различных строк. (Это вариант программы `dup`, которую мы демонстрировали в разделе 1.3). Программа `dedup` использует отображение, ключи которой представляют множество строк, которые уже встречались, и с его помощью обеспечивает отсутствие в выводе дубликатов строк:

gopl.io/ch4/dedup

```
func main() {
    seen := make(map[string]bool) // Множество строк
    input := bufio.NewScanner(os.Stdin)
    for input.Scan() {
        line := input.Text()
        if !seen[line] {
            seen[line] = true
            fmt.Println(line)
        }
    }
    if err := input.Err(); err != nil {
        fmt.Fprintf(os.Stderr, "dedup: %v\n", err)
        os.Exit(1)
    }
}
```

Программисты Go часто описывают отображение, используемое таким образом, как "множество строк", но будьте осторожны: не все значения `map[string]bool` яв-

ляются просто множествами — некоторые из них могут содержать значения `true` и `false`.

Иногда нужно отображение (или множество), ключи которого являются срезами, но поскольку ключи отображений должны быть сравниваемыми, такое отображение не может быть выражено непосредственно. Однако это можно сделать в два этапа. Сначала мы определим вспомогательную функцию `k`, которая отображает каждый ключ на строку, с тем свойством, что `k(x) == k(y)` тогда и только тогда, когда `x` и `y` считаются равными. Затем мы создаем отображение, ключи которого являются строками, применяя вспомогательную функцию к каждому ключу перед обращением к отображению.

В приведенном ниже примере отображение используется для записи количества вызовов `Add` с данным списком строк. Пример использует функцию `fmt.Sprintf` для преобразования среза строк с помощью символов преобразования `%q` в одну строку, которая является подходящим ключом отображения:

```
var m = make(map[string]int)

func k(list []string) string { return fmt.Sprintf("%q", list) }

func Add(list []string) { m[k(list)]++ }
func Count(list []string) int { return m[k(list)] }
```

Такой же подход может использоваться для любого несравниваемого типа ключей, не только для срезов. Иногда этот подход полезен даже для ключей сравниваемых типов, когда требуется определение равенства, отличное от ==, например сравнение без учета регистра для строк. Тип `k(x)` также не обязан быть строкой; подойдут любые сравниваемые типы с необходимым свойством эквивалентности, такие как целые числа, массивы или структуры.

Вот еще один пример отображения в действии: программа, которая подсчитывает количество вхождений каждого различного символа Unicode в его входных данных. Поскольку имеется большое количество всевозможных символов, только небольшая часть которых будет встречаться в любом конкретном документе, отображение является естественным способом отслеживать только те из них, которые встречаются в документе.

gopl.io/ch4/charcount

```
// Charcount вычисляет количество символов Unicode.
package main

import (
    "bufio"
    "fmt"
    "io"
    "os"
    "unicode"
    "unicode/utf8"
)
```

```
func main() {
    counts := make(map[rune]int)    // Количество символов Unicode
    var utflen [utf8.UTFMax + 1]int // Количество длин кодировок UTF-8
    invalid := 0                    // Количество некорректных символов UTF-8

    in := bufio.NewReader(os.Stdin)
    for {
        r, n, err := in.ReadRune() // Возврат руны, байтов, ошибки
        if err == io.EOF {
            break
        }
        if err != nil {
            fmt.Fprintf(os.Stderr, "charcount: %v\n", err)
            os.Exit(1)
        }
        if r == unicode.ReplacementChar && n == 1 {
            invalid++
            continue
        }
        counts[r]++
        utflen[n]++
    }
    fmt.Printf("rune\tcount\n")
    for c, n := range counts {
        fmt.Printf("%q\t%d\n", c, n)
    }
    fmt.Print("\nlen\tcount\n")
    for i, n := range utflen {
        if i > 0 {
            fmt.Printf("%d\t%d\n", i, n)
        }
    }
    if invalid > 0 {
        fmt.Printf("\n%d неверных символов UTF-8\n", invalid)
    }
}
```

Метод `ReadRune` выполняет декодирование UTF-8 и возвращает три значения: декодированную руну, длину ее UTF-8 кода в байтах и значение ошибки. Единственная ошибка, с которой мы ожидаем встречи — это достижение конца файла. Если ввод не является корректным UTF-8-кодом руны, возвращаются руна `unicode.ReplacementChar` и длина 1.

В качестве эксперимента мы применили программу к тексту данной книги[1]. Несмотря на то что он написан на английском языке, в нем все равно имеется ряд сим-

[1] Имеется в виду оригинальное издание книги на английском языке. — *Примеч. пер.*

волов, не являющихся символами ASCII. Вот как выглядела первая пятерка наиболее часто встречавшихся символов:

```
°  27 世  15 界  14  é  13 ˣ 10 ≤ 5 × 5 国 4 4 □ 3
```

Распределение по длинам всех UTF-8-кодов имеет следующий вид:

```
len count
1   765391
2   60
3   70
4   0
```

Тип значения отображения сам может быть составным типом, таким как отображение или срез. В приведенном далее коде типом ключа `graph` является `string`, а типом значения — `map[string]bool`, представляющий множество строк. Концептуально `graph` отображает строку на множество связанных с ней строк, ее преемников в ориентированном графе.

gopl.io/ch4/graph

```
var graph = make(map[string]map[string]bool)

func addEdge(from, to string) {
    edges := graph[from]
    if edges == nil {
        edges = make(map[string]bool)
        graph[from] = edges
    }
    edges[to] = true
}

func hasEdge(from, to string) bool {
    return graph[from][to]
}
```

Функция `addEdge` демонстрирует идиоматический способ отложенного заполнения отображения, при котором инициализация каждого значения выполняется, когда его ключ встречается в первый раз. Функция `hasEdge` показывает, как работает нулевое значение отсутствующей записи отображения: даже если в наличии нет ни `from`, ни `to`, `graph[from][to]` всегда дает значимый результат.

Упражнение 4.8. Измените `charcount` так, чтобы программа подсчитывала количество букв, цифр и прочих категорий Unicode с использованием функций наподобие `unicode.IsLetter`.

Упражнение 4.9. Напишите программу `wordfreq` для подсчета частоты каждого слова во входном текстовом файле. Вызовите `input.Split(bufio.ScanWords)` до первого вызова `Scan` для разбивки текста на слова, а не на строки.

4.4. Структуры

Структура представляет собой агрегированный тип данных, объединяющий нуль или более именованных значений произвольных типов в единое целое. Каждое значение называется *полем*. Классическим примером структуры является запись о сотруднике, полями которой являются уникальный идентификатор, имя сотрудника, его адрес, дата рождения, должность, зарплата и т.п. Все эти поля собраны в единую сущность, которая может копироваться как единое целое, передаваться функции и возвращаться ею, храниться в массивах и т.д.

Приведенные далее инструкции объявляют структурный тип `Employee` и переменную `dilbert`, которая является экземпляром `Employee`:

```
type Employee struct {
    ID         int
    Name       string
    Address    string
    DoB        time.Time
    Position   string
    Salary     int
    ManagerID int
}
var dilbert Employee
```

Доступ к отдельным полям `dilbert` осуществляется с помощью записи с точкой наподобие `dilbert.Name` или `dilbert.DoB`. Поскольку `dilbert` является переменной, ее поля также являются переменными, так что им можно выполнять присваивание:

```
dilbert.Salary -= 5000 // Зарплата снижена, пишет мало строк кода
```

Можно также получать их адреса и доступ посредством этих адресов:

```
position := &dilbert.Position
*position = "Senior " + *position // Повышен в должности
```

Запись с точкой работает и с указателями на структуры:

```
var employeeOfTheMonth *Employee = &dilbert
employeeOfTheMonth.Position += " (активный участник команды)"
```

Последняя инструкция эквивалентна следующей:

```
(*employeeOfTheMonth).Position += " (активный участник команды)"
```

Для заданного уникального идентификатора сотрудника функция `EmployeeByID` возвращает указатель на структуру `Employee`. Для доступа к ее полям можно использовать запись с точкой:

```
func EmployeeByID(id int) *Employee { /* ... */ }

fmt.Println(EmployeeByID(dilbert.ManagerID).Position) // Босс
```

```
id := dilbert.ID
EmployeeByID(id).Salary = 0 // Уволить...
```

Последняя инструкция обновляет структуру `Employee`, на которую указывает результат вызова `EmployeeByID`. Если тип результата `EmployeeByID` изменить на `Employee` вместо `*Employee`, инструкция присваивания не будет компилироваться, поскольку ее левая сторона не будет определять переменную.

Поля обычно записываются по одному в строке; при этом имя поля предшествует его типу. Однако последовательные поля одного и того же типа могут быть объединены, как поля `Name` и `Address` ниже:

```
type Employee struct {
    ID            int
    Name, Address string
    DoB           time.Time
    Position      string
    Salary        int
    ManagerID     int
}
```

Порядок полей имеет важное значение для идентификации типа. Объединим ли мы с упомянутыми полями объявление поля `Position` (также строка) или поменяем местами `Name` и `Address`, мы при этом определим другой тип структуры. Обычно объединяются только объявления связанных полей.

Имя поля структуры экспортируется, если оно начинается с прописной буквы; это основной механизм управления доступом в Go. Структурный тип может содержать комбинацию экспортируемых и неэкспортируемых полей.

Структурные типы, как правило, многословны, потому что зачастую для каждого поля отводится своя строка. Хотя мы могли бы писать весь тип каждый раз, когда это необходимо, повторение такого большого объема утомительно. Вместо этого структурные типы обычно появляются в объявлениях именованных типов, как тип `Employee`.

Именованный структурный тип `S` не может объявить поле того же типа `S`: агрегатное значение не может содержать само себя. (Аналогичное ограничение применимо и к массивам.) Но `S` может объявить поле с типом указателя `*S`, который позволяет нам создавать рекурсивные структуры данных, такие как связанные списки и деревья. Это проиллюстрировано в коде ниже, в котором для реализации сортировки вставками используется бинарное дерево:

gopl.io/ch4/treesort

```
type tree struct {
    value       int
    left, right *tree
}

// Sort сортирует значения "на месте".
func Sort(values []int) {
```

```
    var root *tree
    for _, v := range values {
        root = add(root, v)
    }
    appendValues(values[:0], root)
}

// appendValues добавляет элементы t к values в требуемом
// порядке и возвращает результирующий срез.
func appendValues(values []int, t *tree) []int {
    if t != nil {
        values = appendValues(values, t.left)
        values = append(values, t.value)
        values = appendValues(values, t.right)
    }
    return values
}
func add(t *tree, value int) *tree {
    if t == nil {
        // Эквивалентно возврату &tree{value: value}.
        t = new(tree)
        t.value = value
        return t
    }
    if value < t.value {
        t.left = add(t.left, value)
    } else {
        t.right = add(t.right, value)
    }
    return t
}
```

Нулевое значение для структуры состоит из нулевых значений каждого из ее полей. Обычно желательно, чтобы нулевое значение было естественным или разумным значением по умолчанию. Например, в `bytes.Buffer` начальное значение структуры представляет собой готовый к использованию пустой буфер, а нулевое значение `sync.Mutex`, с которым мы познакомимся в главе 9, "Параллельность и совместно используемые переменные", представляет собой готовый разблокированный мьютекс. Иногда такое разумное первоначальное поведение получается без особых усилий со стороны программиста, но иногда над ним приходится поработать.

Тип структуры без полей называется *пустой структурой*, записываемой как `struct{}`. Она имеет нулевой размер и не несет в себе никакой информации, но может оказаться полезной. Некоторые программисты Go используют ее вместо `bool` в качестве типа значения в отображении, которое представляет собой множество. Такой выбор подчеркивает, что в отображении играют роль только ключи; экономия памяти при этом оказывается незначительной, а синтаксис — более громоздким, поэтому мы, как правило, избегаем такого решения.

```
seen := make(map[string]struct{}) // Множество строк
// ...
if _, ok := seen[s]; !ok {
    seen[s] = struct{}{}
    // ... s встречается впервые ...
}
```

4.4.1. Структурные литералы

Значение структурного типа может быть записано с использованием *структурного литерала*, который определяет значения своих полей:

```
type Point struct{ X, Y int }
p := Point{1, 2}
```

Существуют две разновидности структурных литералов. Первая разновидность, показанная выше, требует указания значения для *каждого* поля в правильном порядке. Это заставляет программиста запоминать точное положение полей в структуре и делает код подверженным ошибкам при последующем добавлении или переупорядочении полей. Так что эта разновидность, как правило, используется только внутри пакета, который определяет структурный тип, или с малыми структурными типами, для которых порядок полей очевиден, например для таких, как `image.Point{x,y}` или `color.RGBA{red,green,blue,alpha}`.

Гораздо чаще используется вторая разновидность, в которой значение структуры инициализируется с помощью перечисления некоторых или всех имен полей с соответствующими значениями, как в приведенной ниже инструкции, взятой из программы Lissajous из раздела 1.4:

```
anim := gif.GIF{LoopCount: nframes}
```

Если поле в таком литерале опущено, оно получает нулевое значение соответствующего типа. Поскольку в литерале указаны имена полей, их порядок становится не имеющим значения.

В одном и том же литерале нельзя смешивать две описанные разновидности. Также нельзя применять первую разновидность литерала (использующую порядок полей), чтобы обойти правило, согласно которому к неэкспортируемым идентификаторам нельзя обращаться из других пакетов.

```
package p
type T struct{ a, b int } // a и b не экспортируемы

package q
import "p"
var _ = p.T{a: 1, b: 2} // Ошибка компиляции: нельзя обращаться к a и b
var _ = p.T{1, 2}       // Ошибка компиляции: нельзя обращаться к a и b
```

Хотя в последней строке выше не упоминаются идентификаторы неэкспортируемых полей, в действительности они используются неявно, поэтому данный код не компилируется.

Значения структур могут быть переданы как аргументы в функцию и быть возвращены из нее. Например, приведенная далее функция масштабирует `Point` с использованием некоторого коэффициента:

```
func Scale(p Point, factor int) Point {
    return Point{p.X * factor, p.Y * factor}
}
fmt.Println(Scale(Point{1, 2}, 5))    // "{5 10}"
```

Для повышения эффективности большие структурные типы обычно передаются в функции или возвращаются из них косвенно с помощью указателя:

```
func Bonus(e *Employee, percent int) int {
    return e.Salary * percent / 100
}
```

Такая передача обязательна, если функция должна модифицировать свой аргумент, поскольку при передаче по значению вызываемая функция получает только копию аргумента, а не ссылку на исходный аргумент.

```
func AwardAnnualRaise(e *Employee) {
    e.Salary = e.Salary * 105 / 100
}
```

Поскольку работа со структурами очень часто выполняется через указатели, можно использовать следующую сокращенную запись для создания и инициализации структурной переменной и получения ее адреса:

```
pp := &Point{1, 2}
```

Эта запись эквивалентна следующей:

```
pp := new(Point)
*pp = Point{1, 2}
```

Однако `&Point{1,2}` можно использовать непосредственно в выражении, таком, как вызов функции.

4.4.2. Сравнение структур

Если все поля структуры сравниваемы, то сравниваема и сама структура, так что два выражения этого типа можно сравнить с помощью оператора == или !=. Операция == поочередно сравнивает соответствующие поля двух структур, так что приведенные ниже выражения сравнения эквивалентны:

```
type Point struct{ X, Y int }

p := Point{1, 2}
```

```
q := Point{2, 1}
fmt.Println(p.X == q.X && p.Y == q.Y) // "false"
fmt.Println(p == q)                   // "false"
```

Сравниваемые структурные типы, как и любые сравниваемые типы, могут использоваться в качестве типа ключа в отображении:

```
type address struct {
    hostname string
    port     int
}
hits := make(map[address]int)
hits[address{"golang.org", 443}]++
```

4.4.3. Встраивание структур и анонимные поля

В этом разделе мы увидим, что необычный механизм *встраивания структур* (struct embedding) позволяет использовать именованный структурный тип в качестве *анонимного поля* (anonymous field) другого структурного типа, обеспечивая удобное синтаксическое сокращение, так что простое выражение с точкой, такое как `x.f`, может означать целую цепочку полей наподобие `x.d.e.f`.

Рассмотрим программу двумерного рисования, которая предоставляет библиотеку фигур, таких как прямоугольники, эллипсы, звезды и колеса. Вот пример двух типов, которые она может определять:

```
type Circle struct {
    X, Y, Radius int
}

type Wheel struct {
    X, Y, Radius, Spokes int
}
```

`Circle` имеет поля для координат центра `X` и `Y`, и `Radius`. `Wheel` обладает всеми возможностями `Circle`, а также имеет поле `Spokes`, содержащее количество радиальных спиц. Давайте создадим колесо:

```
var w Wheel
w.X = 8
w.Y = 8
w.Radius = 5
w.Spokes = 20
```

По мере роста множества фигур вы обязательно заметите сходства и повторения среди них, так что может оказаться удобным выделить их общие части:

```
type Point struct {
    X, Y int
}
```

```
type Circle struct {
    Center Point
    Radius int
}

type Wheel struct {
    Circle Circle
    Spokes int
}
```

Приложение от этого может стать понятнее, но такое изменение усложняет доступ к полям `Wheel`:

```
var w Wheel
w.Circle.Center.X = 8
w.Circle.Center.Y = 8
w.Circle.Radius = 5
w.Spokes = 20
```

Go позволяет объявить поле с типом, но без имени; такие поля называются *анонимными полями*. Тип поля должен представлять собой именованный тип или указатель на именованный тип. Ниже `Circle` и `Wheel` имеют по одному анонимному полю. Мы говорим, что тип `Point` *встроен* (embedded) в тип `Circle`, а `Circle` встроен в `Wheel`.

```
type Circle struct {
    Point
    Radius int
}

type Wheel struct {
    Circle
    Spokes int
}
```

Благодаря встраиванию мы можем обращаться к именам в листьях неявного дерева без указания промежуточных имен:

```
var w Wheel
w.X = 8      // Эквивалентно записи w.Circle.Point.X = 8
w.Y = 8      // Эквивалентно записи w.Circle.Point.Y = 8
w.Radius = 5 // Эквивалентно записи w.Circle.Radius = 5
w.Spokes = 20
```

Явная запись, показанная в комментариях, по-прежнему действительна, однако название "анонимные поля" не совсем верно. Поля `Circle` и `Point` имеют имена, но эти имена необязательны в выражениях с точкой. Мы можем опустить любое или все анонимные поля при выборе их подполей.

К сожалению, не существует соответствующего сокращения для синтаксиса структурного литерала, поэтому ни один из следующих примеров не будет компилироваться:

```
w = Wheel{8, 8, 5, 20}                    // Ошибка: неизвестные поля
w = Wheel{X:8, Y:8, Radius:5, Spokes:20} // Ошибка: неизвестные поля
```

Структурный литерал должен следовать форме объявления типа, поэтому мы должны использовать один из двух приведенных ниже вариантов, которые эквивалентны один другому:

gopl.io/ch4/embed

```
w = Wheel{Circle{Point{8, 8}, 5}, 20}

w = Wheel{
    Circle: Circle{
        Point: Point{X: 8, Y: 8},
        Radius: 5,
    },
    Spokes: 20, // Примечание: завершающая запятая здесь
                // (и после Radius) необходима
}
fmt.Printf("%#v\n", w)
// Вывод:
// Wheel{Circle:Circle{Point:Point{X:8, Y:8}, Radius:5}, Spokes:20}

w.X = 42

fmt.Printf("%#v\n", w)
// Вывод:
// Wheel{Circle:Circle{Point:Point{X:42, Y:8}, Radius:5}, Spokes:20}
```

Обратите внимание на то, как символ # заставляет символы преобразования %v в функции `Printf` выводить значения в виде, похожем на синтаксис Go. Для структур этот вид включает имя каждого поля.

Поскольку "анонимные"' поля имеют неявные имена, у вас не может быть двух анонимных полей одинакового типа, так как их имена будут конфликтовать. И поскольку имя поля неявно определяется его типом, то же самое относится и к видимости поля. В приведенных выше примерах анонимные поля `Point` и `Circle` экспортируются. Если бы они были неэкспортируемыми (`point` и `circle`), мы могли бы по-прежнему использовать сокращенную запись

```
w.X = 8 // Эквивалентно w.circle.point.X = 8
```

Однако явная длинная запись, показанная в комментарии, была бы запрещена вне объявляющего структуру пакета, поскольку `circle` и `point` были бы недоступны.

То, как мы использовали до настоящего момента встраивание структур, является не более чем "синтаксическим сахаром" для выбора полей структуры с помощью записи с точкой. Позже мы увидим, что анонимные поля не обязаны быть структурны-

ми типами; это может быть любой именованный тип или указатель на именованный тип. Но какой смысл встраивать тип, не имеющий подполей?

Ответ имеет отношение к методам. Сокращенная запись, используемая для выбора поля встроенного типа, работает и для выбора его методов. По сути тип внешней структуры получает не только поля встроенного типа, но и его методы. Этот механизм является основным способом получения поведения сложного объекта с помощью композиции более простых. *Композиция* является центральным моментом объектно-ориентированного программирования в Go, и мы будем изучать ее подробнее в разделе 6.3.

4.5. JSON

Запись объектов JavaScript (JavaScript Object Notation — JSON) является стандартной записью для пересылки и получения структурной информации. JSON — не единственный способ такой записи. Аналогичной цели служат XML (раздел 7.14), ASN.1 и Google's Protocol Buffers, и каждый имеет свою нишу; однако из-за простоты, удобочитаемости и всеобщей поддержки наиболее широко используется именно JSON.

Go обладает превосходной поддержкой кодирования и декодирования этих форматов, предоставленной в пакетах стандартной библиотеки `encoding/json`, `encoding/xml`, `encoding/asn1` и других, и все эти пакеты имеют схожие API. В данном разделе приводится краткий обзор наиболее важных частей пакета `encoding/json`.

JSON является кодированием значений JavaScript — строк, чисел, логических значений, массивов и объектов — в виде текста в кодировке Unicode. Это эффективное, но удобочитаемое представление фундаментальных типов данных из главы 3, "Фундаментальные типы данных", и составных типов из этой главы — массивов, срезов, структур и отображений.

Фундаментальными типами JSON являются числа (в десятичной или научной записи), логические значения (`true` или `false`) и строки, которые представляют собой последовательности символов Unicode, заключенные в двойные кавычки, с управляющими последовательностями, начинающимися с обратной косой черты, как в Go, хотя в JSON последовательности `\U`*hhhh* обозначают коды UTF-16, а не руны.

Эти фундаментальные типы могут быть рекурсивно скомбинированы с помощью массивов и объектов JSON. Массив JSON представляет собой упорядоченную последовательность значений, записываемую как список с запятыми в качестве разделителей, заключенный в квадратные скобки; массивы JSON используются для кодирования массивов и срезов Go. Объект JSON является отображением строк на значения, записываемым в виде последовательности пар `name:value`, разделенных запятыми и заключенными в фигурные скобки; объекты JSON используются для кодирования отображений Go (со строковыми ключами) и структур, например:

```
boolean   true
number    273.15
string    "She said \"Hello, 世界\""
array     ["gold", "silver", "bronze"]
```

```
object    {"year": 1980,
           "event": "archery",
           "medals": ["gold", "silver", "bronze"]}
```

Рассмотрим приложение, которое собирает кинообзоры и предлагает рекомендации. Его тип данных `Movie` и типичный список значений объявляются ниже. (Строковые литералы после объявлений полей `Year` и `Color` являются *дескрипторами полей* (field tags); мы объясним их чуть позже.)

```
gopl.io/ch4/movie
type Movie struct {
    Title   string
    Year    int    `json:"released"`
    Color   bool   `json:"color,omitempty"`
    Actors  []string
}

var movies = []Movie{
    {Title: "Casablanca", Year: 1942, Color: false,
        Actors: []string{"Humphrey Bogart", "Ingrid Bergman"}},
    {Title: "Cool Hand Luke", Year: 1967, Color: true,
        Actors: []string{"Paul Newman"}},
    {Title: "Bullitt", Year: 1968, Color: true,
        Actors: []string{"Steve McQueen", "Jacqueline Bisset"}},
    // ...
}
```

Такие структуры данных отлично подходят для JSON и легко конвертируются в обоих направлениях. Преобразование структуры данных Go наподобие `movies` в JSON называется *маршалингом* (marshaling). Маршалинг осуществляется с помощью `json.Marshal`:

```
data, err := json.Marshal(movies)
if err != nil {
    log.Fatalf("Сбой маршалинга JSON: %s", err)
}
fmt.Printf("%s\n", data)
```

`Marshal` генерирует байтовый срез, содержащий очень длинную строку без лишних пробелов; здесь мы разорвали эту строку, чтобы она поместилась на странице книги:

```
[{"Title":"Casablanca","released":1942,"Actors":["Humphrey Bogart","In
grid Bergman"]},{"Title":"Cool Hand Luke","released":1967,"color":true
,"Actors":["Paul Newman"]},{"Title":"Bullitt","released":1968,"color":
true,"Actors":["Steve McQueen","Jacqueline Bisset"]}]
```

Это компактное представление содержит всю необходимую информацию, но его трудно читать. Для чтения человеком можно использовать функцию `json.MarshalIndent`, которая дает аккуратно отформатированное представление с отсту-

пами. Два дополнительных аргумента определяют префикс для каждой строки вывода и строку для каждого уровня отступа:

```
data, err := json.MarshalIndent(movies, "", " ")
if err != nil {
    log.Fatalf("Сбой маршалинга JSON: %s", err)
}
fmt.Printf("%s\n", data)
```

Вывод этого кода имеет следующий вид:

```
[
    {
        "Title": "Casablanca",
        "released": 1942,
        "Actors": [
            "Humphrey Bogart",
            "Ingrid Bergman"
        ]
    },
    {
        "Title": "Cool Hand Luke",
        "released": 1967,
        "color": true,
        "Actors": [
            "Paul Newman"
        ]
    },
    {
        "Title": "Bullitt",
        "released": 1968,
        "color": true,
        "Actors": [
            "Steve McQueen",
            "Jacqueline Bisset"
        ]
    }
]
```

Маршалинг использует имена полей структуры Go в качестве имен полей объектов JSON (используя *рефлексию*, как мы увидим в разделе 12.6). Маршализуются только экспортируемые поля; вот почему мы использовали прописные первые буквы для всех имен полей Go.

Вы могли заметить, что имя поля `Year` изменено в выводе на `released`, а поле `Color` заменено полем `color`. Дело в *дескрипторах полей*. Дескриптор поля представляет собой строку метаданных, связанную с полем структуры во время компиляции:

```
Year int `json:"released"`
Color bool `json:"color,omitempty"`
```

Дескриптор поля может быть любым строковым литералом, но по соглашению он интерпретируется как список пар `key:"value"`, разделенных пробелами; поскольку они содержат двойные кавычки, дескрипторы полей обычно записываются как неформатированные строковые литералы. Ключ `json` управляет поведением пакета `encoding/json` и другими пакетами `encoding/...`, следующими этому соглашению. В первой части дескриптора поля `json` определяется альтернативное имя JSON для поля Go. Дескрипторы полей часто используются для указания идиоматических имен JSON, таких как `total_count` для поля Go с именем `TotalCount`. Дескриптор для `Color` имеет дополнительный параметр, `omitempty`, который указывает, что если поле имеет нулевое значение своего типа (здесь — `false`) или иным образом оказывается пустым, никакого вывода JSON не должно быть. Очевидно, что для черно-белого фильма *Casablanca* JSON-вывод не имеет поля `color`.

Обратная к маршалингу операция, декодирования JSON и заполнения структуры данных Go, называется восстановлением объекта или *демаршалингом* (unmarshaling) и выполняется с помощью `json.Unmarshal`. В приведенном ниже коде данные JSON о фильмах демаршализуются в срез структур, единственным полем которых является `Title`. Путем определения подходящих структур данных Go мы можем выбрать, какие части входных данных JSON будут декодированы, а какие отброшены. При возврате из функции `Unmarshal` она заполняет срез информацией `Title`; все другие имена JSON игнорируются.

```
var titles []struct{ Title string }
if err := json.Unmarshal(data, &titles); err != nil {
    log.Fatalf("Сбой демаршалинга JSON: %s", err)
}
fmt.Println(titles) // "[{Casablanca} {Cool Hand Luke} {Bullitt}]"
```

Многие веб-службы предоставляют интерфейс JSON — вы делаете HTTP-запрос и получаете в ответ нужную информацию в формате JSON. Для иллюстрации давайте запросим GitHub с использованием интерфейса веб-службы. Сначала определим необходимые типы и константы:

gopl.io/ch4/github

```
// Пакет github предоставляет Go API для хостинга GitHub.
// См.https://developer.github.com/v3/search/#searchissues.
package github

import "time"

const IssuesURL = "https://api.github.com/search/issues"

type IssuesSearchResult struct {
    TotalCount int `json:"total_count"`
    Items      []*Issue
}
type Issue struct {
    Number    int
```

```
    HTMLURL   string `json:"html_url"`
    Title     string
    State     string
    User      *User
    CreatedAt time.Time `json:"created_at"`
    Body      string
}
type User struct {
    Login   string
    HTMLURL string `json:"html_url"`
}
```

Как и ранее, имена всех полей структуры должны начинаться с прописной буквы, даже если их имена не являются именами JSON. Однако процесс сопоставления, который связывает имена JSON с именами полей структур Go при демаршалинге не учитывает регистр символов, так что дескриптор поля необходимо использовать только тогда, когда в имени JSON есть знак подчеркивания, которого нет в имени Go. И вновь мы можем отобрать поля для декодирования; ответ GitHub содержит значительно больше информации, чем мы показываем здесь.

Функция `SearchIssues` выполняет HTTP-запрос и декодирует JSON-результат. Поскольку запрос, представленный пользователем, может содержать символы наподобие `?` и `&`, которые имеют специальное значение в URL, мы используем функцию `url.QueryEscape`, чтобы гарантировать, что эти символы будут восприняты буквально.

gopl.io/ch4/github

```
package github

import (
    "encoding/json"
    "fmt"
    "net/http"
    "net/url"
    "strings"
)

// SearchIssues запрашивает GitHub.
func SearchIssues(terms []string) (*IssuesSearchResult, error) {
    q := url.QueryEscape(strings.Join(terms, " "))
    resp, err := http.Get(IssuesURL + "?q=" + q)
    if err != nil {
        return nil, err
    }
    // Необходимо закрыть resp.Body на всех путях выполнения.
    // (В главе 5 вы познакомитесь с более простым решением: 'defer'.)
    if resp.StatusCode != http.StatusOK {
        resp.Body.Close()
        return nil, fmt.Errorf("Сбой запроса: %s", resp.Status)
    }
```

```
    var result IssuesSearchResult
    if err := json.NewDecoder(resp.Body).Decode(&result); err != nil {
        resp.Body.Close()
        return nil, err
    }
    resp.Body.Close()
    return &result, nil
}
```

В предыдущих примерах для декодирования всего содержимого байтового среза как единого JSON-объекта использовалась функция `json.Unmarshal`. Для разнообразия в этом примере используется *потоковый* декодер `json.Decoder`, который может декодировать несколько последовательных объектов JSON из одного и того же потока, хотя нам эта возможность здесь не требуется. Как и следует ожидать, существует соответствующий потоковый кодировщик `json.Encoder`.

Вызов `Decode` заполняет переменную `result`. Существуют различные способы красиво отформатировать ее значение. Простейший продемонстрированный ниже способ — использование текстовой таблицы с фиксированной шириной столбца, но в следующем разделе мы встретимся с более сложным подходом, основанным на шаблонах.

gopl.io/ch4/issues

```
// Выводит таблицу - результат поиска в GitHub.
package main

import (
    "fmt"
    "gopl.io/ch4/github"
    "log"
    "os"
)

func main() {
    result, err := github.SearchIssues(os.Args[1:])
    if err != nil {
        log.Fatal(err)
    }
    fmt.Printf("%d тем:\n", result.TotalCount)
    for _, item := range result.Items {
        fmt.Printf("#%-5d %9.9s %.55s\n",
            item.Number, item.User.Login, item.Title)
    }
}
```

Аргументы командной строки указывают параметры поиска. Показанная ниже команда запрашивает список ошибок среди проектов Go, связанных с декодированием JSON:

```
$ go build gopl.io/ch4/issues
$ ./issues repo:golang/go is:open json decoder
13 тем:
#5680 eaigner encoding/json: set key converter on en/decoder
#6050 gopherbot encoding/json: provide tokenizer
#8658 gopherbot encoding/json: use bufio
#8462 kortschak encoding/json: UnmarshalText confuses json.Unmarshal
#5901 rsc encoding/json: allow override type marshaling
#9812 klauspost encoding/json: string tag not symmetric
#7872 extempora encoding/json: Encoder internally buffers full output
#9650 cespare encoding/json: Decoding gives errPhase when unmarshalin
#6716 gopherbot encoding/json: include field name in unmarshal error me
#6901 lukescott encoding/json, encoding/xml: option to treat unknown fi
#6384 joeshaw encoding/json: encode precise floating point integers u
#6647 btracey x/tools/cmd/godoc: display type kind of each named type
#4237 gjemiller encoding/base64: URLEncoding padding is optional
```

Интерфейс веб-службы GitHub по адресу `https://developer.github.com/v3/` имеет куда больше возможностей, чем мы можем описать в данной книге.

Упражнение 4.10. Измените программу `issues` так, чтобы она сообщала о результатах с учетом их давности, деля на категории, например, поданные менее месяца назад, менее года назад и более года.

Упражнение 4.11. Создайте инструмент, который позволит пользователю создавать, читать, обновлять и закрывать темы GitHub из командной строки, вызывая предпочитаемый пользователем текстовый редактор, когда требуется ввести текст значительного объема.

Упражнение 4.12. Популярный веб-ресурс с комиксами *xkcd* имеет интерфейс JSON. Например, запрос `https://xkcd.com/571/info.0.json` возвращает детальное описание комикса 571, одного из многочисленных фаворитов сайта. Загрузите каждый URL (по одному разу!) и постройте автономный список комиксов. Напишите программу `xkcd`, которая, используя этот список, будет выводить URL и описание каждого комикса, соответствующего условию поиска, заданному в командной строке.

Упражнение 4.13. Веб-служба Open Movie Database `https://omdbapi.com/` на базе JSON позволяет выполнять поиск фильма по названию и загружать его афишу. Напишите инструмент `poster`, который загружает афишу фильма по переданному в командной строке названию.

4.6. Текстовые и HTML-шаблоны

В предыдущем примере используется только простое форматирование, для которого вполне достаточно функции `Printf`. Но иногда форматирование должно быть более сложным, и при этом желательно полностью разделить форматирование и код. Это может быть сделано с помощью пакетов `text/template` и `html/template`, ко-

торые обеспечивают механизм для подстановки значений переменных в текстовый или HTML-шаблон.

Шаблон — это строка (или файл), содержащая один или несколько фрагментов в двойных фигурных скобках, `{{...}}`, именуемых *действиями* (action). Большая часть строки выводится буквально, но действия изменяют поведение программы по умолчанию. Каждое действие содержит выражение на языке шаблонов — простой, но мощной системы записи для вывода значений, выбора полей структуры, вызова функций и методов, выражений управления потоком (например, таких как инструкции `if-else` и циклы по диапазону) и создания других шаблонов. Ниже приведена простая шаблонная строка:

gopl.io/ch4/issuesreport

```
const templ = `{{.TotalCount}} тем:
{{range .Items}}----------------------------------
Number: {{.Number}}
User:   {{.User.Login}}
Title:  {{.Title | printf "%.64s"}}
Age:    {{.CreatedAt | daysAgo}} days
{{end}}`
```

Этот шаблон сначала выводит количество соответствующих запросу тем, а затем номер, имя пользователя, название и возраст в днях каждой из них. В действии имеется понятие текущего значения, которое записывается как точка "`.`". Точка изначально указывает на параметр шаблона, который в данном примере представляет собой `github.IssuesSearchResult`. Действие `{{.TotalCount}}` раскрывается в значение поля `TotalCount`, выводимое обычным способом. Действия `{{range .Items}}` и `{{end}}` создают цикл, так что подстановка последовательных элементов `Items` в текст между этими действиями выполняется несколько раз; точка при этом указывает на очередной элемент `Items`.

В действии запись `|` делает результат одной операции аргументом другой, аналогично конвейеру оболочки Unix. В случае `Title` вторая операция представляет собой функцию `printf`, которая является встроенным синонимом для `fmt.Sprintf` во всех шаблонах. Что касается `Age`, то вторая операция является функцией `daysAgo`, которая преобразует поле `CreatedAt` в количество прошедших дней с использованием `time.Since`:

```
func daysAgo(t time.Time) int {
    return int(time.Since(t).Hours() / 24)
}
```

Обратите внимание, что типом `CreatedAt` является `time.Time`, а не `string`. Так же, как тип может управлять его строковым форматированием (раздел 2.5) путем определения некоторых методов, тип может определять методы для управления его JSON-маршалингом и демаршалингом. Маршалинг JSON значения `time.Time` дает строку в стандартном формате.

Получение вывода с помощью шаблона осуществляется в два этапа. Сначала необходимо выполнить синтаксический анализ шаблона в подходящее внутреннее пред-

ставление, а затем выполнить его для конкретных входных данных. Анализ необходимо выполнить только один раз. Приведенный ниже код создает определенный выше шаблон `templ` и выполняет его синтаксический анализ. Обратите внимание на цепочку вызовов методов: `template.New` создает и возвращает шаблон; `Funcs` добавляет к набору функций, доступных этому шаблону, функцию `daysAgo`, а затем возвращает этот шаблон; и наконец к полученному результату применяется функция `Parse`.

```
report, err := template.New("report").
    Funcs(template.FuncMap{"daysAgo": daysAgo}).
    Parse(templ)
if err != nil {
    log.Fatal(err)
}
```

Поскольку шаблоны обычно фиксированы во время компиляции, сбой синтаксического анализа шаблона указывает на фатальную ошибку в программе. Вспомогательная функция `template.Must` делает обработку ошибок более удобной: она принимает шаблон и ошибку, проверяет, что ошибка является нулевой (в противном случае возникает аварийная ситуация), а затем возвращает шаблон. Мы вернемся к этой идее в разделе 5.9.

После того как шаблон создан, оснащен функцией `daysAgo`, проанализирован и проверен, мы можем выполнить его с использованием `github.IssuesSearchResult` в качестве источника данных и `os.Stdout` в качестве места назначения вывода:

```
var report = template.Must(template.New("issuelist").
    Funcs(template.FuncMap{"daysAgo": daysAgo}).
    Parse(templ))

func main() {
    result, err := github.SearchIssues(os.Args[1:])
    if err != nil {
        log.Fatal(err)
    }
    if err := report.Execute(os.Stdout, result); err != nil {
        log.Fatal(err)
    }
}
```

Эта программа выводит обычный текстовый отчет наподобие следующего:

```
$ go build gopl.io/ch4/issuesreport
$ ./issuesreport repo:golang/go is:open json decoder
13 тем:
--------------------------------
Number:
5680
User: eaigner
Title: encoding/json: set key converter on en/decoder
Age: 750 days
```

```
--------------------------------
Number:
6050
User: gopherbot
Title: encoding/json: provide tokenizer
Age: 695 days
--------------------------------
...
```

Теперь давайте обратимся к пакету `html/template`. Он использует такие же API и язык выражений, как и `text/template`, но добавляет возможности для автоматического и соответствующего контексту экранирования некоторых символов управляющими последовательностями в строках, появляющихся в HTML, JavaScript, CSS или URL. Эти возможности позволяют избежать давнишней проблемы безопасной генерации HTML, *атаки с помощью инъекций*, при которой злоумышленник передает строковое значение наподобие заголовка, содержащее вредоносный код, которое, будучи некорректно преобразовано в шаблоне, предоставляет злоумышленнику контроль над страницей.

Приведенный ниже шаблон выводит список тем в виде таблицы HTML. Обратите внимание на отличающуюся инструкцию `import`:

gopl.io/ch4/issueshtml

```
import "html/template"

var issueList = template.Must(template.New("issuelist").Parse(`
<h1>{{.TotalCount}} тем</h1>
<table>
<tr style='text-align: left'>
  <th>#</th>
  <th>State</th>
  <th>User</th>
  <th>Title</th>
</tr>
{{range .Items}}
<tr>
  <td><a href='{{.HTMLURL}}'>{{.Number}}</a></td>
  <td>{{.State}}</td>
  <td><a href='{{.User.HTMLURL}}'>{{.User.Login}}</a></td>
  <td><a href='{{.HTMLURL}}'>{{.Title}}</a></td>
</tr>
{{end}}
</table>
`))
```

Приведенная ниже команда применяет новый шаблон к результатам немного отличающегося запроса:

```
$ go build gopl.io/ch4/issueshtml
$ ./issueshtml repo:golang/go commenter:gopherbot json encoder>issues.html
```

На рис. 4.4 показано, как выглядит создаваемая таблица в веб-браузере. Ссылки ведут на соответствующие веб-страницы GitHub.

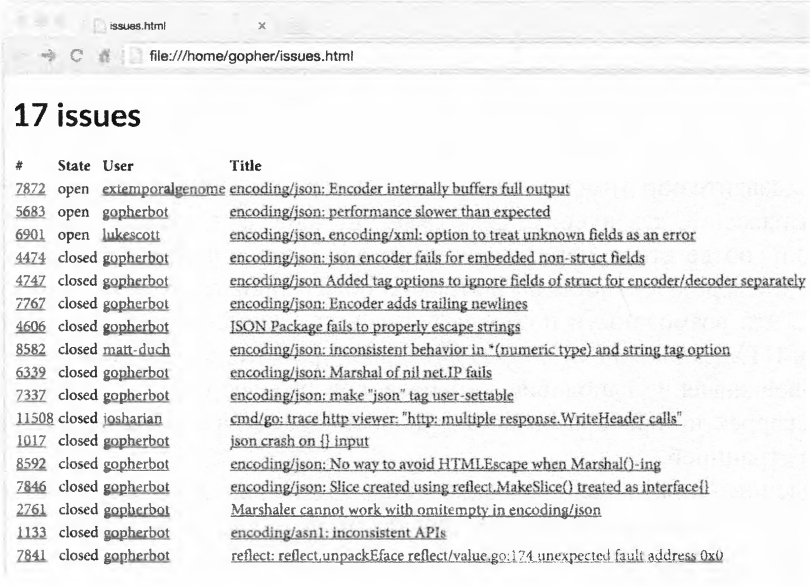


Рис. 4.4. HTML-таблица тем, связанных с JSON-кодированием в Go

Ни одна из тем на рис. 4.4 не представляет собой сложностей для HTML, но мы можем понять, в чем состоит проблема, если рассмотрим темы, названия которых содержат метасимволы HTML, такие как & и <. Для нашего примера мы выбрали две такие темы:

```
$ ./issueshtml repo:golang/go 3133 10535 >issues2.html
```

На рис. 4.5 показан результат этого запроса. Обратите внимание, что пакет `html/template` автоматически преобразует названия тем в HTML так, что они появляются в буквальном виде. Если бы мы по ошибке использовали пакет `text/template`, то четырехсимвольная строка `"&lt;"` была бы выведена как символ `'<'`, а строка `"<link>"` превратилась бы в элемент `link`, изменив структуру HTML-документа и, возможно, сделав его небезопасным.

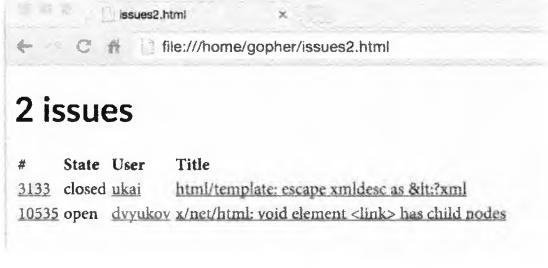


Рис. 4.5. Метасимволы HTML в названиях тем выведены корректно

Такое автоматическое экранирование для полей, содержащих данные HTML, можно подавить с помощью именованного строкового типа `template.HTML`, а не `string`. Аналогичные имена типов имеются для JavaScript, CSS и URL. Приведенная ниже программа демонстрирует использование двух полей с одним и тем же значением, но разными типами: `A` имеет тип `string`, а `B` — `template.HTML`.

gopl.io/ch4/autoescape

```
func main() {
    const templ = `<p>A: {{.A}}</p><p>B: {{.B}}</p>`
    t := template.Must(template.New("escape").Parse(templ))
    var data struct {
        A string        // Обычный текст
        B template.HTML // HTML
    }
    data.A = "<b>Hello!</b>"
    data.B = "<b>Hello!</b>"
    if err := t.Execute(os.Stdout, data); err != nil {
        log.Fatal(err)
    }
}
```

На рис. 4.6 показано, какой вид имеет вывод шаблона в браузере. Как видите, в строке `A` метасимволы были заменены соответствующими управляющими последовательностями для их корректного вывода, а в строке `B` — нет.

Рис. 4.6. Строковые значения экранируются, а значения `template.HTML` — нет

В книге у нас хватает места только для того, чтобы показать лишь основные возможности системы шаблонов. Как всегда, для получения дополнительной информации, обратитесь к документации пакета:

```
$ go doc text/template
$ go doc html/template
```

Упражнение 4.14. Создайте веб-сервер, однократно запрашивающий информацию у GitHub, а затем позволяющий выполнять навигацию по списку сообщений об ошибках, контрольных точек и пользователей.

5

Функции

Функция позволяет "завернуть" последовательность операторов в единое целое, которое может вызываться из других мест в программе, возможно, многократно. Функции позволяют разбить большую работу на более мелкие фрагменты, которые могут быть написаны разными программистами, разделенными во времени и пространстве. Функция скрывает подробности реализации от своих пользователей. По всем этим причинам функции являются важной частью любого языка программирования.

Мы уже видели много функций. Теперь пришло время обсудить их более подробно. Основным примером в этой главе является поисковый агент, т.е. компонент поисковых систем, ответственный за извлечение веб-страниц, выявление ссылок внутри них, извлечение веб-страниц по этим ссылкам и т.д. Поисковый агент обеспечивает широкие возможности для изучения рекурсии, анонимных функций, обработки ошибок и прочих аспектов функций, уникальных для языка программирования Go.

5.1. Объявления функций

Объявление функции имеет имя, список параметров, необязательный список результатов и тело:

```
func имя(список параметров) (список результатов) {
    тело
}
```

Список параметров указывает имена и типы *параметров* функции, которые являются локальными переменными, значения которых, или *аргументы*, поставляются вызывающей функцией. Список результатов указывает типы значений, возвращаемые функцией. Если функция возвращает один неименованный результат или вовсе не возвращает результатов, скобки необязательны и обычно опускаются. Полное отсутствие списка результатов указывает на объявление функции, которая не возвращает никакого значения и вызывается только для выполнения некоторых действий. В функции `hypot`

```
func hypot(x, y float64) float64 {
    return math.Sqrt(x*x + y*y)
```

```
}
fmt.Println(hypot(3, 4)) // "5"
```

x и y являются параметрами в объявлении функции, 3 и 4 — аргументами вызова, а возвращает функция значение `float64`.

Подобно параметрам, результаты могут быть именованными. В этом случае каждое имя объявляет локальную переменную, инициализируемую нулевым значением ее типа.

Функция, которая имеет список результатов, должна заканчиваться оператором `return`, если только исполнение явно не может дойти до конца функции, например если функция заканчивается вызовом `panic` или бесконечным циклом `for` без `break`.

Как мы видели в функции `hypot`, последовательность параметров или результатов одного и того же типа может быть факторизована, т.е. записана так, что сам тип записывается только однократно. Следующие два объявления эквивалентны:

```
func f(i, j, k int, s, t string) { /* ... */ }
func f(i int, j int, k int, s string, t string) { /* ... */ }
```

Далее представлены четыре способа объявления функции с двумя параметрами и одним результатом, имеющими тип `int`. Пустой идентификатор может использоваться для того, чтобы подчеркнуть, что этот параметр не используется.

```
func add(x int, y int) int   { return x + y }
func sub(x, y int) (z int)   { z = x - y; return }
func first(x int, _ int) int { return x }
func zero(int, int) int      { return 0 }

fmt.Printf("%T\n", add)      // "func(int, int) int"
fmt.Printf("%T\n", sub)      // "func(int, int) int"
fmt.Printf("%T\n", first)    // "func(int, int) int"
fmt.Printf("%T\n", zero)     // "func(int, int) int"
```

Тип функции иногда называют ее *сигнатурой*. Две функции имеют один и тот же тип, или сигнатуру, если они имеют одну и ту же последовательность типов параметров и одну и ту же последовательность типов результатов. Имена параметров и результатов не влияют на тип функции, так же как не влияет то, было ли объявление функции факторизовано.

Каждый вызов функции должен предоставить аргумент для каждого параметра функции в том порядке, в котором были объявлены эти параметры. В Go отсутствует как концепция параметров по умолчанию, так и какие-либо способы для указания аргументов по имени, так что имена параметров и результатов не имеют никакого значения для вызывающего кода, кроме как для целей документирования.

Параметры являются локальными переменными в теле функции; их начальные значения равны аргументам, переданным вызывающей функцией. Параметры функции и именованные результаты являются переменными в том же лексическом блоке, что и наиболее внешние локальные переменные функции.

Аргументы передаются в функции *по значению*, поэтому функция получает копию каждого аргумента; изменения в копии не влияют на исходный объект. Однако если аргумент содержит некоторую ссылку, как, например, указатель, срез, отображение, функцию или канал, то функция может влиять на объекты вызывающего кода с помощью *косвенного* внесения изменений с использованием аргумента-ссылки.

Иногда вы можете столкнуться с объявлением функции без тела, указывающим, что эта функция реализована на языке, отличном от языка Go. Такое объявление определяет сигнатуру функции.

```
package math
func Sin(x float64) float64 // Реализована на ассемблере
```

5.2. Рекурсия

Функции могут быть рекурсивными, т.е. могут вызывать сами себя прямо или косвенно. Рекурсия является мощным инструментом для решения множества задач и, конечно, необходима для обработки рекурсивных структур данных. В разделе 4.4 мы использовали рекурсию по дереву для реализации простой сортировки вставкой. В этом разделе мы снова будем использовать ее для обработки HTML-документов.

В приведенном ниже примере программы используется нестандартный пакет `golang.org/x/net/html`, который предоставляет синтаксический анализатор HTML. Репозитории `golang.org/x/...` хранят пакеты, разработанные и поддерживаемые командой Go для приложений, таких как сети, обработка текста на разных языках, мобильные платформы, манипуляции изображениями, криптография и инструменты разработчика. Эти пакеты не входят в стандартную библиотеку, потому что они все еще находятся в стадии разработки или оказываются редко нужными большинству программистов на Go.

Необходимые нам части API `golang.org/x/net/html` показаны ниже. Функция `html.Parse` считывает последовательность байтов, выполняет ее синтаксический анализ и возвращает корень дерева документа HTML, который имеет тип `html.Node`. HTML имеет узлы нескольких разновидностей — текст, комментарии и т.д., — но сейчас мы сосредоточим свое внимание только на узлах *элементов* вида `<name key='value'>`.

golang.org/x/net/html

```
package html

type Node struct {
    Type                    NodeType
    Data                    string
    Attr                    []Attribute
    FirstChild, NextSibling *Node
}

type NodeType int32
```

```
const (
    ErrorNode NodeType = iota
    TextNode
    DocumentNode
    ElementNode
    CommentNode
    DoctypeNode
)

type Attribute struct {
    Key, Val string
}

func Parse(r io.Reader) (*Node, error)
```

Функция `main` выполняет синтаксический анализ данных со стандартного входа как HTML, извлекая ссылки с использованием рекурсивной функции `visit`, и выводя каждую найденную ссылку:

gopl.io/ch5/findlinks1

```
// Findlinks1 выводит ссылки в HTML-документе,
// прочитанном со стандартного входа.
package main

import (
    "fmt"
    "os"

    "golang.org/x/net/html"
)

func main() {
    doc, err := html.Parse(os.Stdin)
    if err != nil {
        fmt.Fprintf(os.Stderr, "findlinks1: %v\n", err)
        os.Exit(1)
    }
    for _, link := range visit(nil, doc) {
        fmt.Println(link)
    }
}
```

Функция `visit` обходит дерево узлов HTML, извлекает ссылки из атрибута `href` каждого элемента `<a href='...'>`, добавляет ссылки в срез строк и возвращает результирующий срез:

```
// visit добавляет в links все ссылки,
// найденные в n, и возвращает результат.
func visit(links []string, n *html.Node) []string {
```

```
    if n.Type == html.ElementNode && n.Data == "a" {
        for _, a := range n.Attr {
            if a.Key == "href" {
                links = append(links, a.Val)
            }
        }
    }
    for c := n.FirstChild; c != nil; c = c.NextSibling {
        links = visit(links, c)
    }
    return links
}
```

Для спуска по дереву для узла `n` функция `visit` рекурсивно вызывает себя для каждого из дочерних узлов `n`, которые хранятся в связанном списке `FirstChild`.

Давайте запустим `findlinks` для начальной страницы Go, передав выход программы `fetch` (раздел 1.5) на вход `findlinks`. Для краткости мы немного отредактировали получающийся вывод:

```
$ go build gopl.io/ch1/fetch
$ go build gopl.io/ch5/findlinks1
$ ./fetch https://golang.org | ./findlinks1
#
/doc/
/pkg/
/help/
/blog/
http://play.golang.org/
//tour.golang.org/
https://golang.org/dl/
//blog.golang.org/
/LICENSE
/doc/tos.html
http://www.google.com/intl/en/policies/privacy/
```

Обратите внимание на разнообразие видов ссылок, которые появляются на странице. Позже мы увидим, как разрешать их относительно базового URL, `https://golang.org`, чтобы получать абсолютные URL.

Приведенная далее программа использует рекурсию по дереву узлов HTML для вывода наброска структуры дерева; когда программа встречает каждый элемент, она помещает дескриптор элемента в стек, а затем выводит стек:

gopl.io/ch5/outline

```
func main() {
    doc, err := html.Parse(os.Stdin)
    if err != nil {
        fmt.Fprintf(os.Stderr, "outline: %v\n", err)
        os.Exit(1)
    }
```

```
    outline(nil, doc)
}
func outline(stack []string, n *html.Node) {
    if n.Type == html.ElementNode {
        stack = append(stack, n.Data) // Внесение дескриптора в стек
        fmt.Println(stack)
    }
    for c := n.FirstChild; c != nil; c = c.NextSibling {
        outline(stack, c)
    }
}
```

Обратите внимание на одну тонкость: хотя `outline` "заталкивает" элемент в `stack`, соответствующей функции для снятия элемента со стека нет. Когда функция `outline` рекурсивно вызывает саму себя, вызываемая функция получает копию `stack`. Хотя вызываемая функция может добавлять элементы в этот срез, изменяя базовый массив, и, возможно, даже выделяет новый массив, она не изменяет исходные элементы, видимые вызывающей функции. Поэтому при возврате из функции `stack` вызывающей функции остается таким же, как и до вызова.

Вот как выглядит набросок `https://golang.org` (вновь отредактированный для краткости):

```
$ go build gopl.io/ch5/outline
$ ./fetch https://golang.org | ./outline
[html]
[html head]
[html head meta]
[html head title]
[html head link]
[html body]
[html body div]
[html body div]
[html body div div]
[html body div div form]
[html body div div form div]
[html body div div form div a]
...
```

Как вы можете увидеть, экспериментируя с функцией `outline`, большинство документов HTML могут быть обработаны только с несколькими уровнями рекурсии; однако не так уж трудно создать патологические веб-страницы, которые потребуют чрезвычайно глубокой рекурсии.

Многие реализации языка программирования используют стек вызовов функций фиксированного размера; типичные размеры колеблются от 64 Кбайт до 2 Мбайт. Фиксированный размер стека ограничивает глубину рекурсии, так что надо быть осторожным, чтобы избежать *переполнения стека* при рекурсивном обходе больших структур данных; фиксированный размер стека может даже создать угрозу безопасности. К счастью, типичные реализации Go используют стеки переменного размера,

которые, начинаясь с малого размера, могут вырастать по мере необходимости до гигабайтных величин.

Это позволяет безопасно использовать рекурсию и не беспокоиться о переполнении стека.

Упражнение 5.1. Измените программу `findlinks` так, чтобы она обходила связанный список `n.FirstChild` с помощью рекурсивных вызовов `visit`, а не с помощью цикла.

Упражнение 5.2. Напишите функцию для заполнения отображения, ключами которого являются имена элементов (`p`, `div`, `span` и т.д.), а значениями — количество элементов с таким именем в дереве HTML-документа.

Упражнение 5.3. Напишите функцию для вывода содержимого всех текстовых узлов в дереве документа HTML. Не входите в элементы `<script>` и `<style>`, поскольку их содержимое в веб-браузере не является видимым.

Упражнение 5.4. Расширьте функцию `visit` так, чтобы она извлекала другие разновидности ссылок из документа, такие как изображения, сценарии и листы стилей.

5.3. Множественные возвращаемые значения

Функция может возвращать более одного результата. Мы видели много примеров таких многозначных функций из стандартных пакетов, которые возвращают два значения, желаемый результат вычислений и значение ошибки или логическое значение, указывающее, были ли вычисления корректно выполнены. В следующем примере показано, как написать собственную функцию с несколькими возвращаемыми значениями.

Приведенная ниже программа является версией программы `findlinks`, которая сама выполняет запрос HTTP, так что нам больше не надо запускать программу `fetch`. Поскольку и запрос HTTP, и синтаксический анализ могут быть неудачными, функция `findLinks` объявляет два результата: список обнаруженных ссылок и ошибку. Кстати, HTML-анализатор обычно может восстановиться при некорректных входных данных и построить документ, содержащий узлы ошибок, так что функция `Parse` редко бывает неудачной; когда такое происходит, это, как правило, связано с ошибками ввода-вывода.

gopl.io/ch5/findlinks2

```
func main() {
    for _, url := range os.Args[1:] {
        links, err := findLinks(url)
        if err != nil {
            fmt.Fprintf(os.Stderr, "findlinks2: %v\n", err)
            continue
        }
        for _, link := range links {
            fmt.Println(link)
        }
    }
}
```

```
// findLinks выполняет HTTP-запрос GET для заданного url,
// выполняет синтаксический анализ ответа как HTML-документа
// и извлекает и возвращает ссылки.
func findLinks(url string) ([]string, error) {
    resp, err := http.Get(url)
    if err != nil {
        return nil, err
    }
    if resp.StatusCode != http.StatusOK {
        resp.Body.Close()
        return nil, fmt.Errorf("получение %s: %s", url, resp.Status)
    }
    doc, err := html.Parse(resp.Body)
    resp.Body.Close()
    if err != nil {
        return nil, fmt.Errorf("анализ %s как HTML: %v", url, err)
    }
    return visit(nil, doc), nil
}
```

В функции `findLinks` имеются четыре оператора `return`, каждый из которых возвращает пару значений. Первые три заставляют функцию передавать информацию об ошибке из пакетов `http` и `html` вызывающей функции. В первом случае ошибка возвращается без изменений; во втором и третьем она дополняется контекстной информацией с помощью функции `fmt.Errorf` (раздел 7.8). Если вызов `findLinks` является успешным, последний оператор `return` возвращает срез ссылок без указания ошибки.

Мы должны гарантировать закрытие `resp.Body`, чтобы даже в случае ошибки сетевые ресурсы были освобождены должным образом. Сборщик мусора Go освобождает неиспользуемую память, но не считайте, что он будет освобождать неиспользуемые ресурсы операционной системы, такие как открытые файлы и сетевые подключения. Они должны быть закрыты явным образом.

Результат вызова многозначной функции представляет собой кортеж значений. При вызове такой функции необходимо явным образом присваивать значения переменным, если любая из них должна быть использована:

```
links, err := findLinks(url)
```

Чтобы проигнорировать одно из значений, присвойте его пустому идентификатору:

```
links, _ := findLinks(url) // Ошибки проигнорированы
```

Результат многозначного вызова сам может быть получен из вызова многозначной функции, как в приведенном ниже примере функции, которая ведет себя как `findLinks`, но записывает свой аргумент:

```
func findLinksLog(url string) ([]string, error) {
    log.Printf("findLinks %s", url)
    return findLinks(url)
}
```

Многозначный вызов может являться единственным аргументом при вызове функции с несколькими параметрами. Хотя такое редко используется в промышленном коде, эта возможность иногда оказывается удобной во время отладки, так как позволяет нам выводить все результаты вызова с помощью одной инструкции. Приведенные далее два выражения вывода действуют одинаково:

```
log.Println(findLinks(url))
links, err := findLinks(url)
log.Println(links, err)
```

Тщательно подобранные имена могут документировать смысл результатов функции. Имена являются особенно важными, когда функция возвращает несколько результатов одного и того же типа, как, например

```
func Size(rect image.Rectangle) (width, height int)
func Split(path string) (dir, file string)
func HourMinSec(t time.Time) (hour, minute, second int)
```

Однако не всегда необходимо именовать несколько результатов исключительно для документирования. Например, по соглашению последний результат типа `bool` указывает успешное завершение функции; результат `error` часто не требует никаких пояснений.

В функции с именованными результатами операнды оператора `return` могут быть опущены. Это называется *пустым возвратом* (bare return).

```
// CountWordsAndImages выполняет HTTP-запрос GET HTML-документа
// url и возвращает количество слов и изображений в нем.
func CountWordsAndImages(url string) (words, images int, err error) {
    resp, err := http.Get(url)
    if err != nil {
        return
    }
    doc, err := html.Parse(resp.Body)
    resp.Body.Close()
    if err != nil {
        err = fmt.Errorf("parsing HTML: %s", err)
        return
    }
    words, images = countWordsAndImages(doc)
    return
}
func countWordsAndImages(n *html.Node)(words,images int){ /* ... */ }
```

Пустой возврат представляет собой сокращенную запись для возврата всех именованных результирующих переменных в указанном в объявлении порядке, так что в приведенной выше функции каждая инструкция `return` эквивалентна записи

```
return words, images, err
```

В функциях, подобных этой, в которых имеется много операторов `return` и несколько результатов, пустой возврат может уменьшить дублирование кода, но он редко облегчает понимание исходного текста. Например, на первый взгляд не очевидно, что два первых возврата эквивалентны `return 0,0,err` (поскольку результирующие переменные `words` и `images` инициализируются нулевыми значениями соответствующего типа) и что окончательная инструкция `return` эквивалентна записи `return words,images,nil`. По этой причине пустой возврат лучше всего использовать экономно.

Упражнение 5.5. Реализуйте функцию `countWordsAndImages` (см. разделение на слова в упр. 4.9).

Упражнение 5.6. Модифицируйте функцию `corner` из `gopl.io/ch3/surface` (раздел 3.2), чтобы она использовала именованные результаты и инструкцию пустого возврата.

5.4. Ошибки

Одни функции всегда успешно решают свои задачи. Например, `strings.Contains` и `strconv.FormatBool` имеют четко определенные результаты для всех возможных значений аргументов и не могут завершиться неудачно — за исключением непредсказуемых и катастрофических сценариев наподобие нехватки памяти, когда симптом проявляется далеко от причины и практически нет надежд на восстановление.

Другие функции всегда успешны, пока выполняются их предусловия. Например, функция `time.Date` всегда создает `time.Time` из его компонентов (год, месяц и т.д.), если только последний аргумент (часовой пояс) не равен `nil`; в этом случае возникает аварийная ситуация. Такая аварийная ситуация — верный признак ошибки в вызывающем коде и никогда не должна возникать в хорошо написанной программе.

Для многих других функций даже в хорошо написанных программах успех не гарантируется, потому что он зависит от факторов, находящихся вне контроля программиста. Например, любая функция, выполняющая ввод-вывод, должна учитывать возможность ошибки, и только наивный программист верит в то, что простое чтение или запись никогда не сбоит. Когда наиболее надежные операции неожиданно оказываются неудачными, обязательно необходимо знать, в чем дело.Таким образом, ошибки являются важной частью API пакета или пользовательского интерфейса приложения, и сбой является лишь одним из нескольких ожидаемых поведений программы. В этом заключается подход Go к обработке ошибок.

Функция, для которой отказ является ожидаемым поведением, возвращает дополнительный результат, обычно последний. Если сбой имеет только одну возможную причину, результат представляет собой булево значение, обычно с именем `ok`, как в приведенном ниже примере поиска в кеше, который всегда выполняется успешно, если только в кеше имеется запись для искомого ключа:

```
value, ok := cache.Lookup(key)
if !ok {
```

```
    // ...cache[key] не существует...
}
```

Чаще всего, и в особенности для ввода-вывода, неудачи могут иметь множество разнообразных причин, так что вызывающая функция должна знать, что именно произошло. В таких случаях тип дополнительного результата — `error`.

Встроенный тип `error` является типом интерфейса. Мы узнаем, что это означает и какие следствия для обработки ошибок из этого вытекают, в главе 7, "Интерфейсы". Сейчас же достаточно знать, что ошибка может быть нулевой или не нулевой, что нулевая ошибка означает успешное завершение и что ненулевая ошибка содержит строку сообщения об ошибке, которую мы можем получить путем вызова ее метода `Error` или путем вывода с помощью `fmt.Println(err)` или `fmt.Printf("%v",err)`.

Обычно, когда функция возвращает ненулевую ошибку, ее результаты не определены и должны быть проигнорированы. Однако некоторые функции при ошибке могут возвращать частичные результаты. Например, если произошла ошибка во время чтения из файла, вызов `Read` возвращает количество байтов, которые функция смогла прочитать, *и* ошибку, описывающую возникшую проблему. Для обеспечения правильного поведения некоторым вызывающим функциям может потребоваться обработать неполные данные до обработки ошибки, поэтому важно, чтобы такие функции точно документировали свои результаты.

Подход Go отличает его от многих других языков программирования, в которых об отказах сообщается с помощью механизма *исключений*, а не обычных значений. Хотя в Go имеется определенный механизм исключений, как мы увидим в разделе 5.9, он используется только для действительно неожиданных неприятностей, указывающих на наличие серьезной ошибки, а не на обычные ошибки подпрограмм, которые надежная программа должна ожидать и обрабатывать.

Причина такого подхода заключается в том, что исключения, как правило, смешивают описание ошибки с обрабатывающим ее потоком управления, что часто приводит к нежелательным результатам: обычные ошибки подпрограмм предоставляются конечному пользователю в виде малопонятного состояния стека с полной информацией о структуре программы, но при этом недостаточно четкой информацией о том, что именно пошло не так.

Go же, напротив, для реакции на ошибки использует обычные механизмы потока управления наподобие инструкции `if` или `return`. Этот стиль, несомненно, требует уделять больше внимания логике обработки ошибок, но ясность и точность обработки ошибок того стоят.

5.4.1. Стратегии обработки ошибок

Когда вызов функции возвращает ошибку, вызывающая функция должна проверить ее и принять соответствующие меры. В зависимости от ситуации может быть ряд возможных действий вызывающей функции. Давайте рассмотрим пять из них.

Первый и наиболее распространенный случай — *распространение* ошибки, так что сбой в подпрограмме становится сбоем вызывающей функции. Мы видели

примеры этого в функции `findLinks` из раздела 5.3. Если вызов `http.Get` неудачен, `findLinks` передает ошибку HTTP вызывающей функции без дальнейших церемоний:

```
resp, err := http.Get(url)
if err != nil {
    return nil, err
}
```

В противоположность этому, если неудачен вызов `html.Parse`, функция `findLinks` не возвращает ошибку HTML-анализатора непосредственно, потому что в ней не хватает двух важнейших частей информации: того, что ошибка произошла в синтаксическом анализаторе, и URL документа, анализ которого выполнялся. В этом случае функция `findLinks` создает новое сообщение об ошибке, которое включает в себя эту информацию, а также базовую ошибку синтаксического анализатора:

```
doc, err := html.Parse(resp.Body)
resp.Body.Close()
if err != nil {
    return nil, fmt.Errorf("Анализ %s как HTML: %v", url, err)
}
```

Функция `fmt.Errorf` форматирует сообщение об ошибке, используя функцию `fmt.Sprintf`, и возвращает новое значение `error`. Мы используем ее для создания описательной ошибки с помощью добавления информации о контексте к исходному сообщению об ошибке. Когда ошибка в конечном итоге обрабатывается в функции `main` программы, должна быть предоставлена четкая причинно-следственная цепочка от источника, чем-то напоминающая расследование несчастного случая в NASA:

```
Источник: авария: нет парашюта: ошибка переключателя: сбой реле
```

Поскольку сообщения об ошибках часто соединены в одну цепочку, строки сообщения не стоит начинать с прописных букв, а также следует избегать символов новой строки. Получаемые сообщения могут быть длинными, зато будут самодостаточны при поиске с помощью таких инструментов, как `grep`.

При разработке сообщений об ошибках будьте осмотрительны, чтобы каждое из них было осмысленным и достаточно подробным описанием проблемы; они должны быть последовательными и согласованными, чтобы ошибки, возвращаемые одной и той же функцией или группой функций в одном пакете, были похожи по форме и могли рассматриваться одним и тем же способом.

Например, пакет `os` гарантирует, что каждая ошибка, возвращенная файловой операцией, такой как `os.Open`, или методами `Read`, `Write` или `Close` открытого файла, не только описывает характер сбоя (отсутствие прав доступа, нет такого каталога и т.д.), но и содержит имя файла, так что вызывающей функции не нужно включать эту информацию в сообщение об ошибке, которое она создает.

В общем случае вызов `f(x)` несет ответственность за предоставление информации о предпринятой операции `f` и значении аргумента `x`, если они относятся к контексту ошибки. Вызывающая функция отвечает за добавление дальнейшей информации,

которой обладает она, но не функция `f(x)`, например URL в вызове `html.Parse` из приведенного выше исходного текста.

Давайте перейдем ко второй стратегии для обработки ошибок. Для ошибок, которые представляют преходящие или непредсказуемые проблемы, может иметь смысл *повторить* сбойную операцию, возможно, с задержкой между попытками, и, вероятно, с ограничением, накладываемым на количество попыток или на время, которое можно затратить на выполнение попыток.

gopl.io/ch5/wait

```
// WaitForServer пытается соединиться с сервером заданного URL.
// Попытки предпринимаются в течение минуты с растущими интервалами.
// Сообщает об ошибке, если все попытки неудачны.
func WaitForServer(url string) error {
    const timeout = 1 * time.Minute
    deadline := time.Now().Add(timeout)
    for tries := 0; time.Now().Before(deadline); tries++ {
        _, err := http.Head(url)
        if err == nil {
            return nil // Успешное соединение
        }
        log.Printf("Сервер не отвечает (%s); повтор...", err)
        time.Sleep(time.Second << uint(tries)) // Увеличение задержки
    }
    return fmt.Errorf("Сервер %s не отвечает; время %s ", url, timeout)
}
```

Далее, если прогресс невозможен, вызывающая функция может вывести сообщение об ошибке и элегантно завершить выполнение программы, но этот способ действий, как правило, зарезервирован для основного пакета программы. Библиотечные функции обычно должны передавать ошибки вызывающей функции, кроме случаев, когда ошибка является признаком внутренней несогласованности.

```
// (В функции main.)
if err := WaitForServer(url); err != nil {
    fmt.Fprintf(os.Stderr, "Сервер не работает: %v\n", err)
    os.Exit(1)
}
```

Более удобным способом достижения того же эффекта является вызов `log.Fatalf`. Как и все функции пакета `log`, по умолчанию она предваряет сообщение об ошибке текущими временем и датой.

```
if err := WaitForServer(url); err != nil {
    log.Fatalf("Сервер не работает: %v\n", err)
}
```

Формат по умолчанию полезен при работе "долгоиграющего" сервера и менее удобен для интерактивного инструмента:

```
2006/01/02 15:04:05 Сервер не работает: неверный домен: bad.gopl.io
```

Более привлекательный вывод можно получить, задав префикс, используемый в пакете `log`, и подавив отображение даты и времени:

```
log.SetPrefix("wait: ")
log.SetFlags(0)
```

В некоторых случаях достаточно просто записать ошибку в журнал и продолжить работу, возможно, с уменьшенной функциональностью. При этом вновь можно выбирать между использованием пакета `log`, который добавляет обычный префикс:

```
if err := Ping(); err != nil {
    log.Printf("ошибка ping: %v; сеть отключена", err)
}
```

и выводом непосредственно в стандартный поток ошибок:

```
if err := Ping(); err != nil {
    fmt.Fprintf(os.Stderr, "ошибка ping: %v; сеть отключена\n", err)
}
```

(Все функции пакета `log` добавляют символ новой строки, если таковой отсутствует.)

Последняя стратегия — в редких случаях можно безопасно полностью игнорировать ошибку:

```
dir, err := ioutil.TempDir("", "scratch")
if err != nil {
    return fmt.Errorf("ошибка создания временного каталога: %v", err)
}
// ... используем временный каталог ...
os.RemoveAll(dir) // Игнорируем ошибки; $TMPDIR периодически очищается
```

Вызов `os.RemoveAll` может завершиться неудачно, но программа это игнорирует, потому что операционная система сама периодически очищает временный каталог. В этом случае игнорирование ошибки было преднамеренным, но логика программы выглядит так, как будто мы просто забыли обработать ошибку. Выработайте привычку проверять ошибки после каждого вызова функции, а если вы преднамеренно игнорируете их в каком-то месте, явно документируйте свои намерения.

Обработка ошибок в Go имеет свой "ритм". После проверки сначала обычно обрабатывается ошибка, а уже затем — успешное выполнение. Если сбой приводит к выходу из функции, успешное продолжение работы выполняется не в блоке `else`, а на более высоком уровне, с меньшим отступом. Функции, как правило, демонстрируют общую структуру с рядом предварительных проверок, чтобы избежать ошибок, после чего следует основной код функции с минимальным отступом.

5.4.2. Конец файла (EOF)

Обычно различные ошибки, которые может возвращать функция, интересны конечному пользователю, но не для промежуточной логики программы. Иногда, однако, программы должны предпринимать различные действия в зависимости от типа

происшедшей ошибки. Рассмотрим попытки чтения *n* байтов данных из файла. Если *n* — длина файла, любой сбой представляет собой ошибку. С другой стороны, если вызывающая функция пытается многократно читать блоки фиксированного размера до конца файла, то она должна реагировать на ошибку достижения конца файла иначе, чем на все прочие ошибки. По этой причине пакет `io` гарантирует, что о невозможности чтения, вызванной достижением конца файла, всегда сообщается как об отдельной ошибке `io.EOF`, которая определена следующим образом:

```
package io
import "errors"
// EOF – это ошибка, возвращаемая функцией Read,
// когда больше нет данных для чтения.
var EOF = errors.New("EOF")
```

Вызывающая функция может обнаружить это условие с помощью простого сравнения, как ниже показано в цикле, который читает руны из стандартного ввода. (В программе `charcount` из раздела 4.3 представлен более полный исходный текст.)

```
in := bufio.NewReader(os.Stdin)
for {
    r, _, err := in.ReadRune()
    if err == io.EOF {
        break // Чтение завершено
    }
    if err != nil {
        return fmt.Errorf("сбой чтения: %v", err)
    }
    // ...использование r...
}
```

Поскольку условие конца файла не содержит никакой информации, о которой могло бы сообщить, `io.EOF` имеет фиксированное сообщение об ошибке — `"EOF"`. Для других ошибок может потребоваться более полный отчет, так сказать, более качественная и количественная информация, поэтому у них нет фиксированного значения ошибок. В разделе 7.11 мы представим более систематический способ, как отличать определенные значения ошибок от прочих.

5.5. Значения-функции

Функции являются *значениями первого класса* в Go: подобно другим значениям, значения-функции имеют типы, и они могут быть присвоены переменным или, например, переданы в функцию или возвращены из нее. Значение-функция может быть вызвано подобно любой другой функции, например:

```
func square(n int) int     { return n * n }
func negative(n int) int   { return -n }
func product(m, n int) int { return m * n }
```

```
f := square
fmt.Println(f(3))     // "9"

f = negative
fmt.Println(f(3))     // "-3"
fmt.Printf("%T\n", f) // "func(int) int"

f = product           // Ошибка компиляции: нельзя присваивать
                      // func(int, int) int переменной func(int) int
```

Нулевым значением типа функции является `nil`. Вызов нулевой функции приводит к аварийной ситуации:

```
var f func(int) int
f(3)   // Аварийная ситуация: вызов nil-функции
```

Значение-функцию можно сравнить с `nil`:

```
var f func(int) int
if f != nil {
    f(3)
}
```

Однако сами значения-функции являются не сравниваемыми одно с другим, так что их нельзя сравнивать одно с другим или использовать в качестве ключей в отображении.

Значения-функции позволяют параметризировать наши функции не только данными, но и поведением. Стандартные библиотеки содержат много примеров тому. Например, `strings.Map` применяет функцию к каждому символу строки, объединяя результаты в другую строку:

```
func add1(r rune) rune { return r + 1 }

fmt.Println(strings.Map(add1, "HAL-9000")) // "IBM.:111"
fmt.Println(strings.Map(add1, "VMS"))      // "WNT"
fmt.Println(strings.Map(add1, "Admix"))    // "Benjy"
```

Функция `findLinks` из раздела 5.2 использует вспомогательную функцию, `visit`, чтобы посетить все узлы в документе HTML и применить действие к каждому из них. С помощью значения-функции мы можем отделить логику обхода дерева от логики действия, применяемого к каждому узлу, что позволит нам выполнять обход с различными действиями:

gopl.io/ch5/outline2

```
// forEachNode вызывает функции pre(x) и post(x) для каждого узла x
// в дереве с корнем n. Обе функции необязательны.
// pre вызывается до посещения дочерних узлов, а post – после.
func forEachNode(n *html.Node, pre, post func(n *html.Node)) {
    if pre != nil {
        pre(n)
    }
```

```
    for c := n.FirstChild; c != nil; c = c.NextSibling {
        forEachNode(c, pre, post)
    }
    if post != nil {
        post(n)
    }
}
```

Функция `forEachNode` принимает два аргумента-функции: один — для вызова перед посещением дочерних узлов, а второй — для вызова после такого посещения. Это обеспечивает высокую степень гибкости для вызывающей функции. Например, функции `startElement` и `endElement` выводят открывающий и закрывающий дескрипторы HTML-элемента наподобие `<b>...</b>`:

```
var depth int

func startElement(n *html.Node) {
    if n.Type == html.ElementNode {
        fmt.Printf("%*s<%s>\n", depth*2, "", n.Data)
        depth++
    }
}

func endElement(n *html.Node) {
    if n.Type == html.ElementNode {
    depth--
    fmt.Printf("%*s</%s>\n", depth*2, "", n.Data)
    }
}
```

Эти функции также обеспечивают отступы с помощью еще одного трюка функции `fmt.Printf`. Символ `*` в `%*s` выводит строку, дополненную переменным количеством пробелов. Ширина вывода и выводимая строка переданы как аргументы `depth*2` и `""`.

При вызове `forEachNode` с HTML-документом следующим образом:

```
forEachNode(doc, startElement, endElement)
```

мы получим более красивый вывод нашей рассмотренной ранее программы `outline`:

```
$ go build gopl.io/ch5/outline2
$ ./outline2 http://gopl.io
<html>
  <head>
    <meta>
    </meta>
    <title>
    </title>
    <style>
    </style>
```

```
  </head>
  <body>
    <table>
      <tbody>
        <tr>
          <td>
            <a>
              <img>
              </img>
...
```

Упражнение 5.7. Разработайте `startElement` и `endElement` для обобщенного вывода HTML. Выводите узлы комментариев, текстовые узлы и атрибуты каждого элемента (`<a href='...'>`). Используйте сокращенный вывод наподобие `<img/>` вместо `<img></img>`, когда элемент не имеет дочерних узлов. Напишите тестовую программу, чтобы убедиться в корректности выполняемого анализа (см. главу 11, "Тестирование".)

Упражнение 5.8. Измените функцию `forEachNode` так, чтобы функции `pre` и `post` возвращали булево значение, указывающее, следует ли продолжать обход дерева. Воспользуйтесь ими для написания функции `ElementByID` с приведенной ниже сигнатурой, которая находит первый HTML-элемент с указанным атрибутом `id`. Функция должна прекращать обход дерева, как только соответствующий элемент найден:

```
func ElementByID(doc *html.Node, id string) *html.Node
```

Упражнение 5.9. Напишите функцию `expand(s string, f func(string) string) string`, которая заменяет каждую подстроку `"$foo"` в `s` текстом, который возвращается вызовом `f("foo")`.

5.6. Анонимные функции

Именованные функции могут быть объявлены только на уровне пакета, но мы можем использовать *литерал функции* для обозначения значения-функции в любом выражении. Литерал функции записывается как объявление функции, но без имени после ключевого слова `func`. Это выражение, и его значение называется *анонимной функцией*.

Литерал функции позволяет определить функцию в точке использования. В качестве примера приведенный выше вызов `strings.Map` можно переписать следующим образом:

```
strings.Map(func(r rune) rune { return r + 1 }, "HAL9000")
```

Что более важно, определенная таким образом функция имеет доступ ко всему лексическому окружению, так что внутренняя функция может обращаться к переменным их охватывающей функции, как показано в примере ниже:

gopl.io/ch5/squares

```
// squares возвращает функцию, которая при каждом вызове
// возвращает квадрат очередного числа.
func squares() func() int {
    var x int
    return func() int {
        x++
        return x * x
    }
}

func main() {
    f := squares()
    fmt.Println(f()) //  "1"
    fmt.Println(f()) //  "4"
    fmt.Println(f()) //  "9"
    fmt.Println(f()) // "16"
}
```

Функция `squares` возвращает другую функцию, типа `func() int`. Вызов `squares` создает локальную переменную `x` и возвращает анонимную функцию, которая при каждом вызове увеличивает `x` и возвращает ее квадрат. Второй вызов `squares` создаст вторую переменную `x` и вернет новую анонимную функцию, которая будет увеличивать эту переменную.

В примере `squares` продемонстрировано, что значения-функции не только являются кодом, но могут иметь и состояние. Внутренняя анонимная функция может обращаться к локальным переменным охватывающей функции `squares` и обновлять их. Эти скрытые ссылки на переменные являются причиной, по которой мы классифицируем функции как ссылочные типы и по которой значения-функции не являются сравниваемыми. Значения-функции, подобные приведенной, реализуются с помощью технологии под названием *замыкания*, и программисты Go часто используют этот термин для значений-функций.

Здесь мы опять видим пример того, что время жизни переменной не определяется ее областью видимости: переменная `x` существует после того, как функция `squares` возвращает управление функции `main`, несмотря на то что `x` скрыта внутри `f`.

В качестве несколько академического примера анонимных функций рассмотрим проблему вычисления последовательности курсов информатики, которая удовлетворяет требованиям каждого из них, заключающимся в том, что определенный курс опирается на другие курсы, которые должны быть изучены до него. Условия приведены в таблице `prereqs` ниже, которая представляет собой отображение каждого курса на список курсов, которые должны быть пройдены до данного курса.

gopl.io/ch5/toposort

```
// prereqs отображает каждый курс на список курсов, которые
// должны быть прочитаны раньше него.
var prereqs = map[string][]string{
    "algorithms": {"data structures"},
```

```
    "calculus"  : {"linear algebra"},
    "compilers" : {
                    "data structures",
                    "formal languages",
                    "computer organization",
                  },
    "data structures"       : {"discrete math"},
    "databases"             : {"data structures"},
    "discrete math"         : {"intro to programming"},
    "formal languages"      : {"discrete math"},
    "networks"              : {"operating systems"},
    "operating systems"     : {"data structures",
                               "computer organization"},
    "programming languages": {"data structures",
                               "computer organization"},
}
```

Такого рода задача известна как топологическая сортировка. Концептуально информация о необходимых курсах формирует ориентированный граф с узлом для каждого курса и ребрами от него к курсам, от которых он зависит. Этот граф ацикличен: не существует пути от курса, который вел бы обратно к этому курсу. Вычислить допустимую последовательность курсов можно с помощью поиска в графе в глубину с использованием следующего кода:

```
func main() {
    for i, course := range topoSort(prereqs) {
        fmt.Printf("%d:\t%s\n", i+1, course)
    }
}

func topoSort(m map[string][]string) []string {
    var order []string
    seen := make(map[string]bool)
    var visitAll func(items []string)
    visitAll = func(items []string) {
        for _, item := range items {
            if !seen[item] {
                seen[item] = true
                visitAll(m[item])
                order = append(order, item)
            }
        }
    }

    var keys []string
    for key := range m {
        keys = append(keys, key)
    }
```

```
    sort.Strings(keys)
    visitAll(keys)
    return order
}
```

Когда анонимная функция требует рекурсию, как показано в данном примере, мы должны сначала объявить переменную, а затем присвоить ей анонимную функцию. Если эти два шага объединить в один в объявлении, литерал функции не будет находиться в области видимости переменной `visitAll`, так что он не будет иметь возможности вызывать себя рекурсивно:

```
visitAll := func(items []string) {
    // ...
    visitAll(m[item]) // Ошибка компиляции: не определена visitAll
    // ...
}
```

Ниже приведен вывод программы `toposort`. Он детерминирован — это часто требуемое свойство, которое не всегда удается получить бесплатно. Здесь значениями отображения `prereqs` являются срезы, а не отображения, так что порядок их итераций детерминирован, и мы сортируем ключи `prereqs` до первого вызова `visitAll`.

```
 1:    intro to programming
 2:    discrete math
 3:    data structures
 4:    algorithms
 5:    linear algebra
 6:    calculus
 7:    formal languages
 8:    computer organization
 9:    compilers
10:    databases
11:    operating systems
12:    networks
13:    programming languages
```

Давайте вернемся к нашему примеру `findLinks`. Мы перенесли функцию извлечения ссылок `links.Extract` в собственный пакет, поскольку мы вновь будем использовать ее в главе 8, "Go-подпрограммы и каналы". Мы заменили функцию `visit` анонимной функцией, которая выполняет добавление в срез `links` непосредственно и использует `frEachNode` для обработки обхода. Поскольку функция `Extract` нуждается только в функции `pre`, в качестве аргумента `post` он передает значение `nil`.

gopl.io/ch5/links

```
// Пакет links предоставляет функцию для извлечения ссылок.
package links

import (
    "fmt"
    "golang.org/x/net/html"
```

```
    "net/http"
)

// Extract выполняет HTTP-запрос GET по определенному URL, выполняет
// синтаксический анализ HTML и возвращает ссылки в HTML-документе.
func Extract(url string) ([]string, error) {
    resp, err := http.Get(url)
    if err != nil {
        return nil, err
    }
    if resp.StatusCode != http.StatusOK {
        resp.Body.Close()
        return nil, fmt.Errorf("получение %s: %s", url, resp.Status)
    }
    doc, err := html.Parse(resp.Body)
    resp.Body.Close()
    if err != nil {
        return nil, fmt.Errorf("анализ %s как HTML: %v", url, err)
    }
    var links []string
    visitNode := func(n *html.Node) {
        if n.Type == html.ElementNode && n.Data == "a" {
            for _, a := range n.Attr {
                if a.Key != "href" {
                    continue
                }
                link, err := resp.Request.URL.Parse(a.Val)
                if err != nil {
                    continue // Игнорируем некорректные URL
                }
                links = append(links, link.String())
            }
        }
    }
    forEachNode(doc, visitNode, nil)
    return links, nil
}
```

Вместо добавления необработанного значения атрибута `href` в срез `links` данная версия анализирует его как URL-адрес относительно базового URL документа, `resp.Request.URL`. Результирующая ссылка `link` получается в абсолютной форме, пригодная для использования в вызове `http.Get`.

Обход веб — по сути своей задача обхода графа. В примере `topoSort` показан обход в глубину; для нашего веб-сканера мы будем использовать обход в ширину, по крайней мере сначала. В главе 8, "Go-подпрограммы и каналы", мы рассмотрим параллельный обход.

Приведенная ниже функция инкапсулирует суть обхода в ширину. Вызывающая функция предоставляет исходный список элементов для посещения `worklist` и зна-

чение-функцию f, вызываемую для каждого элемента. Каждый элемент идентифицируется строкой. Функция f возвращает список новых элементов для добавления в рабочий список. Функция breadthFirst возвращает управление после посещения всех элементов. Она поддерживает множество строк для гарантии того, что ни один элемент не будет посещен дважды.

gopl.io/ch5/findlinks3

```
// breadthFirst вызывает f для каждого элемента в worklist.
// Все элементы, возвращаемые f, добавляются в worklist.
// f вызывается для каждого элемента не более одного раза.
func breadthFirst(f func(item string) []string, worklist []string) {
    seen := make(map[string]bool)
    for len(worklist) > 0 {
        items := worklist
        worklist = nil
        for _, item := range items {
            if !seen[item] {
                seen[item] = true
                worklist = append(worklist, f(item)...)
            }
        }
    }
}
```

Как мы объясняли в главе 3, "Фундаментальные типы данных", аргумент "f(item)..." приводит к тому, что все элементы в списке, возвращенные f, добавляются в worklist.

В нашем сканере элементы являются URL. Функция crawl, которую мы передаем breadthFirst, выводит URL, извлекает из него ссылки и возвращает их так, что они тоже оказываются посещенными:

```
func crawl(url string) []string {
    fmt.Println(url)
    list, err := links.Extract(url)
    if err != nil {
        log.Print(err)
    }
    return list
}
```

Для запуска сканера мы используем аргументы командной строки в качестве начальных URL.

```
func main() {
    // Поиск в ширину, начиная с аргумента командной строки.
    breadthFirst(crawl, os.Args[1:])
}
```

Давайте начнем сканирование с адреса https://golang.org. Вот некоторые из возвращенных ссылок:

```
$ go build gopl.io/ch5/findlinks3
$ ./findlinks3 https://golang.org
https://golang.org/
https://golang.org/doc/
https://golang.org/pkg/
https://golang.org/project/
https://code.google.com/p/gotour/
https://golang.org/doc/code.html
https://www.youtube.com/watch?v=XCsL89YtqCs
http://research.swtch.com/gotour
https://vimeo.com/53221560
...
```

Процесс завершается, когда все доступные веб-страницы будут просканированы или при исчерпании памяти компьютера.

Упражнение 5.10. Перепишите `topoSort` так, чтобы вместо срезов использовались отображения, и устраните начальную сортировку. Убедитесь, что результаты, пусть и недетерминированные, представляют собой корректную топологическую сортировку.

Упражнение 5.11. Преподаватель линейной алгебры (linear algebra) считает, что до его курса следует прослушать курс матанализа (calculus). Перепишите функцию `topoSort` так, чтобы она сообщала о наличии циклов.

Упражнение 5.12. Функции `startElement` и `endElement` в `gopl.io/ch5/outline2` (раздел 5.5) совместно используют глобальную переменную `depth`. Превратите их в анонимные функции, которые совместно используют локальную переменную функции `outline`.

Упражнение 5.13. Модифицируйте функцию `crawl` так, чтобы она делала локальные копии найденных ею страниц, при необходимости создавая каталоги. Не делайте копии страниц, полученных из других доменов. Например, если исходная страница поступает с адреса `golang.org`, сохраняйте все страницы оттуда, но не сохраняйте страницы, например, с `vimeo.com`.

Упражнение 5.14. Используйте функцию `breadthFirst` для исследования другой структуры. Например, вы можете использовать зависимости учебных курсов из примера `topoSort` (ориентированный граф), иерархию файловой системы на своем компьютере (дерево) или список маршрутов автобусов в своем городе (неориентированный граф).

5.6.1. Предупреждение о захвате переменных итераций

В этом разделе мы рассмотрим ловушку правил лексической области видимости Go, которая может привести к неожиданным результатам. Мы настоятельно рекомендуем вам разобраться в этой проблеме, прежде чем продолжить работу, потому что в эту ловушку попадают даже опытные программисты.

Рассмотрим программу, которая должна создать набор каталогов, а затем удалить их. Мы можем использовать срез значений-функций для хранения операций по очистке. (Для краткости в этом примере мы опустили любую обработку ошибок.)

```
var rmdirs []func()
for _, d := range tempDirs() {
    dir := d                // Примечание: необходимо!
    os.MkdirAll(dir, 0755) // Создание родительских каталогов
    rmdirs = append(rmdirs, func() {
        os.RemoveAll(dir)
    })
}

// ...do some work...

for _, rmdir := range rmdirs {
    rmdir()                 // Очистка
}
```

Вы можете удивиться, почему мы присваиваем переменную цикла d новой локальной переменной `dir` в теле цикла, вместо того чтобы просто дать имя `dir` переменной цикла, но это будет некорректным решением:

```
var rmdirs []func()
for _, dir := range tempDirs() {
    os.MkdirAll(dir, 0755)
    rmdirs = append(rmdirs, func() {
        os.RemoveAll(dir)   // Примечание: неверно!
    })
}
```

Причина заключается в следствии из правил области видимости для переменных цикла. В приведенном выше исходном тексте цикла `for` вводит новые лексический блок, в котором объявлена переменная `dir`. Все значения-функции, созданные этим циклом, "захватывают" и совместно используют одну и ту же переменную — адресуемое место в памяти, а не его значение в этот конкретный момент. Поэтому к тому моменту, когда во время очистки вызываются соответствующие функции, переменная `dir` несколько раз оказывается обновленной завершенным к этому моменту циклом `for`. Таким образом, `dir` хранит значение из последней итерации, а следовательно, все вызовы `os.RemoveAll` попытаются удалить один и тот же каталог.

Зачастую для решения этой проблемы создается внутренняя переменная (в нашем примере — `dir`) с тем же именем, что и у внешней переменной, копией которой она является. Это приводит к странно выглядящему, но ключевому объявлению переменной наподобие следующего:

```
for _, dir := range tempDirs() {
    dir := dir // Объявление внутренней dir, инициализированной
               // значением внешней dir
    // ...
}
```

Этот риск не является чем-то уникальным, характерным только для цикла по диапазону. Цикл в примере ниже сталкивается с той же проблемой из-за непреднамеренного захвата индексной переменной `i`.

```
var rmdirs []func()
dirs := tempDirs()
for i := 0; i < len(dirs); i++ {
    os.MkdirAll(dirs[i], 0755) // OK
    rmdirs = append(rmdirs, func() {
        os.RemoveAll(dirs[i]) // Примечание: неверно!
    })
}
```

Проблема захвата переменной итерации чаще всего встречается при использовании инструкции `go` (глава 8, "Go-подпрограммы и каналы") или `defer` (с которой мы вскоре познакомимся), поскольку обе они могут задержать выполнение значения-функции до момента после завершения цикла. Но эта проблема не является проблемой, свойственной именно `go` или `defer`.

5.7. Вариативные функции

Вариативная функция (функция с переменным количеством аргументов) — это функция, которая может быть вызвана с разным количеством аргументов. Наиболее знакомым примером является функция `fmt.Printf` и ее разновидности. Функция `Printf` требует одного фиксированного аргумента в начале, после которого может следовать любое количество других аргументов.

Чтобы объявить вариативную функцию, перед типом последнего параметра указывается троеточие "`...`", которое говорит о том, что функция может быть вызвана с любым количеством аргументов данного типа:

gopl.io/ch5/sum

```
func sum(vals ...int) int {
    total := 0
    for _, val := range vals {
        total += val
    }
    return total
}
```

Показанная выше функция `sum` возвращает сумму нуля или более аргументов типа `int`. В теле функции типом `vals` является срез `[]int`. При вызове функции `sum` в качестве параметра `vals` может быть предоставлено любое количество значений:

```
fmt.Println(sum())           // "0"
fmt.Println(sum(3))          // "3"
fmt.Println(sum(1, 2, 3, 4)) // "10"
```

Неявно вызывающая функция выделяет память для массива, копирует в него аргументы и передает в функцию срез, представляющий весь массив. Последний вызов, показанный выше, таким образом, ведет себя так же, как приведенный ниже, в котором показано, как вызывать вариативные функции, когда аргументы уже находятся в срезе: следует добавить троеточие после последнего аргумента:

```
values := []int{1, 2, 3, 4}
fmt.Println(sum(values...)) // "10"
```

Хотя параметр `...int` ведет себя в теле функции, как срез, тип вариативной функции отличается от типа функции с параметром, являющимся обычным срезом:

```
func f(...int) {}
func g([]int) {}

fmt.Printf("%T\n", f) // "func(...int)"
fmt.Printf("%T\n", g) // "func([]int)"
```

Вариативные функции часто используются для форматирования строк. Показанная далее функция `errorf` создает форматированное сообщение об ошибке с номером строки в его начале. Суффикс `f` является широко распространенным соглашением об именовании вариативных функций, которые принимают строку формата в стиле `Printf`.

```
func errorf(linenum int, format string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, "Стр. %d: ", linenum)
    fmt.Fprintf(os.Stderr, format, args...)
    fmt.Fprintln(os.Stderr)
}

linenum, name := 12, "count"
errorf(linenum, "не определен %s", name) // "Стр. 12: не определен count"
```

Тип `interface{}` означает, что данная функция может принимать любые значения в качестве последних аргументов, как будет пояснено в главе 7, "Интерфейсы".

Упражнение 5.15. Напишите вариативные функции `max` и `min`, аналогичные функции `sum`. Что должны делать эти функции, будучи вызванными без аргументов? Напишите варианты функций, требующие как минимум одного аргумента.

Упражнение 5.16. Напишите вариативную версию функции `strings.Join`.

Упражнение 5.17. Напишите вариативную функцию `ElementsByTagName`, которая для данного дерева узла HTML и нуля или нескольких имен возвращает все элементы, которые соответствуют одному из этих имен. Вот два примера вызова такой функции:

```
func ElementsByTagName(doc *html.Node, name ...string) []*html.Node
images := ElementsByTagName(doc, "img")
headings := ElementsByTagName(doc, "h1", "h2", "h3", "h4")
```

5.8. Отложенные вызовы функций

Наши примеры `findLinks` использовали выходные данные `http.Get` как входные данные `html.Parse`. Это хорошо работает, если содержание запрашиваемого URL действительно представляет собой HTML, но многие страницы содержат изображения, текст и другие форматы файлов. Передача таких файлов HTML-анализатору может иметь нежелательные последствия.

Приведенная ниже программа извлекает документ HTML и выводит его название. Функция `title` проверяет заголовок `Content-Type` ответа сервера и возвращает ошибку, если документ не является документом HTML.

gopl.io/ch5/title1

```
func title(url string) error {
    resp, err := http.Get(url)
    if err != nil {
        return err
    }

    // Проверка, что заголовок Content-Type представляет собой HTML
    // (например, "text/html; charset=utf-8").
    ct := resp.Header.Get("Content-Type")
    if ct != "text/html" && !strings.HasPrefix(ct, "text/html;") {
        resp.Body.Close()
        return fmt.Errorf("%s имеет тип %s, не text/html", url, ct)
    }

    doc, err := html.Parse(resp.Body)
    resp.Body.Close()
    if err != nil {
        return fmt.Errorf("анализ %s как HTML: %v", url, err)
    }

    visitNode := func(n *html.Node) {
        if n.Type == html.ElementNode && n.Data == "title" &&
            n.FirstChild != nil {
            fmt.Println(n.FirstChild.Data)
        }
    }
    forEachNode(doc, visitNode, nil)
    return nil
}
```

Вот типичная (слегка отредактированная) сессия:

```
$ go build gopl.io/ch5/title1
$ ./title1 http://gopl.io
The Go Programming Language
$ ./title1 https://golang.org/doc/effective_go.html
Effective Go - The Go Programming Language
```

```
$ ./title1 https://golang.org/doc/gopher/frontpage.png
title: https://golang.org/doc/gopher/frontpage.png
имеет тип image/png, не text/html
```

Обратите внимание на дублированный вызов `resp.Body.Close()`, который гарантирует, что `title` закрывает сетевое подключение на всех путях выполнения, в том числе при сбоях. По мере того как функции становятся более сложными и должны обрабатывать больше ошибок, такое дублирование логики очистки может стать проблемой при поддержке программы. Давайте посмотрим, как новый механизм Go `defer` позволяет упростить решение этой проблемы.

Синтаксически инструкция `defer` является обычным вызовом функции или метода, которому предшествует ключевое слово `defer`. Функция и выражения аргументов вычисляются при выполнении инструкции, но фактический вызов *откладывается* до завершения функции, содержащей инструкцию `defer`, независимо от того, как она завершается: обычным способом с помощью оператора `return` или в результате аварийной ситуации. Может быть отложено любое количество вызовов; они выполняются в порядке, обратном тому, в котором они были отложены.

Инструкция `defer` часто используется с такими парными операциями, как открытие и закрытие, подключение и отключение или блокировка и разблокирование — для гарантии освобождения ресурсов во всех случаях, независимо от того, насколько сложен поток управления. Правильное место инструкции `defer`, которая освобождает ресурс, — сразу же после того, как ресурс был успешно захвачен. В функции `title` ниже один отложенный вызов заменяет оба предыдущих вызова `resp.Body.Close()`:

gopl.io/ch5/title2

```
func title(url string) error {
    resp, err := http.Get(url)
    if err != nil {
        return err
    }
    defer resp.Body.Close()

    ct := resp.Header.Get("Content-Type")
    if ct != "text/html" && !strings.HasPrefix(ct, "text/html;") {
        return fmt.Errorf("%s имеет тип %s, не text/html", url, ct)
    }

    doc, err := html.Parse(resp.Body)
    if err != nil {
        return fmt.Errorf("анализ %s как HTML: %v", url, err)
    }

    // ...вывод элемента title документа...

    return nil
}
```

Тот же шаблон может использоваться для других ресурсов, помимо сетевых подключений, например, чтобы закрыть открытый файл:

io/ioutil
```
package ioutil

func ReadFile(filename string) ([]byte, error) {
    f, err := os.Open(filename)
    if err != nil {
        return nil, err
    }
    defer f.Close()
    return ReadAll(f)
}
```

или для разблокирования мьютекса (раздел 9.2):

```
var mu sync.Mutex
var m = make(map[string]int)

func lookup(key string) int {
    mu.Lock()
    defer mu.Unlock()
    return m[key]
}
```

Инструкция `defer` может также использоваться для пары отладочных записей о входе в некоторую функцию и выходе из нее. Показанная ниже функция `BigSlowOperation` немедленно вызывает функцию `trace`, которая выполняет запись о входе в функцию и возвращает значение-функцию, которая при вызове выполняет запись о выходе из функции. Таким образом, с помощью отложенного вызова возвращаемой функции мы можем выполнять запись о входе в функцию и выходе из нее в одной инструкции и даже передавать между этими двумя действиями значения, например время начала работы функции. Но не забывайте о завершающей паре скобок в инструкции `defer`, иначе "входное" действие будет выполнено на выходе из функции, а "выходное" не будет выполнено вовсе!

gopl.io/ch5/trace
```
func bigSlowOperation() {
    defer trace("bigSlowOperation")() // Не забывайте о скобках!
    // ... длительная работа ...
    time.Sleep(10 * time.Second)      // Имитация долгой работы
}

func trace(msg string) func() {
    start := time.Now()
    log.Printf("вход в %s", msg)
    return func() { log.Printf("выход из %s (%s)", msg, time.
Since(start)) }
}
```

При каждом вызове `bigSlowOperation` выполняется запись о времени входа в нее и выхода, а также о времени работы данной функции (мы использовали `time.Sleep` для имитации длительной работы функции):

```
$ go build gopl.io/ch5/trace
$ ./trace
2015/11/18 09:53:26 вход в bigSlowOperation
2015/11/18 09:53:36 выход из bigSlowOperation (10.000589217s)
```

Отложенные функции выполняются *после* того, как инструкция возврата обновляет переменные результатов функции. Поскольку анонимная функция может обращаться к переменным охватывающей функции, включая именованные результаты, отложенная анонимная функция имеет доступ к результатам функции, в которой вызвана.

Рассмотрим функцию `double`:

```
func double(x int) int {
    return x + x
}
```

Присвоив имя ее результирующей переменной и добавив инструкцию `defer`, мы можем заставить функцию выводить свои аргументы и результат при каждом вызове:

```
func double(x int) (result int) {
    defer func() { fmt.Printf("double(%d) = %d\n", x, result) }()
    return x + x
}

_ = double(4)
// Вывод:
// "double(4) = 8"
```

Этот трюк является излишеством для такой простой функции, как `double`, но может быть полезен в функции со многими операторами `return`.

Отложенная анонимная функция может даже изменять значения, которые возвращает охватывающая функция:

```
func triple(x int) (result int) {
    defer func() { result += x }()
    return double(x)
}

fmt.Println(triple(4)) // "12"
```

Поскольку отложенные функции не выполняются до самого конца выполнения функции, инструкция `defer` в цикле заслуживает дополнительного изучения. У приведенного ниже кода могут возникнуть проблемы из-за исчерпания доступных файловых дескрипторов, поскольку ни один файл не будет закрыт, пока не будут обработаны все файлы:

```
for _, filename := range filenames {
    f, err := os.Open(filename)
```

```
    if err != nil {
        return err
    }
    defer f.Close() // Примечание: рискованно; может привести
                    // к исчерпанию файловых дескрипторов
    // ...работа с f...
}
```

Одним из решений может быть перенос тела цикла, включая инструкцию `defer`, в другую функцию, которая вызывается на каждой итерации.

```
for _, filename := range filenames {
    if err := doFile(filename); err != nil {
        return err
    }
}

func doFile(filename string) error {
    f, err := os.Open(filename)
    if err != nil {
        return err
    }
    defer f.Close()
    // ...работа с f...
}
```

Приведенный ниже пример представляет собой усовершенствованную программу `fetch` (раздел 1.5), которая записывает HTTP-ответ в локальный файл, а не в стандартный вывод. Она выводит имя файла из последнего компонента пути URL, который получает с помощью функции `path.Base`.

gopl.io/ch5/fetch

```
// Fetch загружает URL и возвращает имя и длину локального файла.

func fetch(url string) (filename string, n int64, err error) {
    resp, err := http.Get(url)
    if err != nil {
        return "", 0, err
    }
    defer resp.Body.Close()
    local := path.Base(resp.Request.URL.Path)
    if local == "/" {
        local = "index.html"
    }
    f, err := os.Create(local)
    if err != nil {
        return "", 0, err
    }
    n, err = io.Copy(f, resp.Body)
```

```
    // Закрытие файла; если есть ошибка Copy, возвращаем ее.
    if closeErr := f.Close(); err == nil {
        err = closeErr
    }
    return local, n, err
}
```

Отложенный вызов `resp.Body.Close` уже должен быть вам знаком. Есть большой соблазн использовать второй отложенный вызов, `f.Close`, чтобы закрыть локальный файл, но это будет неправильно, потому что `os.Create` открывает файл для записи, создавая его при необходимости. Но во многих файловых системах, в особенности NFS, ошибки записи не фиксируются немедленно и могут быть отложены до тех пор, пока файл не будет закрыт. Невозможность проверить результат операции закрытия может привести к тому, что потеря данных останется незамеченной. Однако если и `io.Copy`, и `f.Close` завершаются неудачно, следует предпочесть отчет об ошибке `io.Copy`, поскольку она произошла первой и, скорее всего, сообщит главную причину неприятностей.

Упражнение 5.18. Перепишите, не изменяя ее поведение, функцию `fetch` так, чтобы она использовала `defer` для закрытия записываемого файла.

5.9. Аварийная ситуация

Система типов Go отлавливает множество ошибок во время компиляции, но другие ошибки, такие как обращение за границами массива или разыменование нулевого указателя, требуют проверок времени выполнения. Когда среда выполнения Go обнаруживает эти ошибки, мы сталкиваемся с аварийной ситуацией[1].

Во время типичной аварийной ситуации нормальное выполнение программы останавливается, выполняются все отложенные вызовы функций в go-подпрограмме и программа аварийно завершает работу с записью соответствующего сообщения. Это журнальное сообщение содержит *значение аварийной ситуации*, которое обычно представляет собой некоторое сообщение об ошибке, и, для каждой go-подпрограммы, *трассировку стека*, показывающую состояние стека вызовов функций, которые были активны во время аварийной ситуации. Этого журнального сообщения часто оказывается достаточно, чтобы диагностировать причину проблемы без повторного запуска программы, поэтому его всегда следует включать в передаваемый разработчику отчет о найденных ошибках.

Не все аварийные ситуации возникают из-за ошибок времени выполнения. Встроенная функция `panic` может вызываться непосредственно; в качестве аргумента она принимает любое значение. Такой вызов — зачастую наилучшее, что вы можете сделать, когда случается некоторая "невозможная" ситуация, например выполнение достигает `case` в инструкции `switch`, которого согласно логике программы достичь не может:

[1] В языке Go для описания таких ситуаций используется понятие *"паника" (panic)*. — *Примеч. пер.*

```
switch s := suit(drawCard()); s {
case "Spades":    // ...
case "Hearts":    // ...
case "Diamonds":  // ...
case "Clubs":     // ...
default:
panic(fmt.Sprintf("неверная карта %q", s)) // Джокер?
}
```

Хорошей практикой является проверка выполнения предусловий функции, но такие проверки могут легко оказаться избыточными. Если только вы не можете предоставить более информативное сообщение об ошибке или обнаружить ошибку заранее, нет смысла в проверке, которую среда выполнения осуществит сама:

```
func Reset(x *Buffer) {
    if x == nil {
        panic("x is nil") // Нет смысла!
    }
    x.elements = nil
}
```

Хотя механизм аварийных ситуаций Go напоминает исключения в других языках программирования, ситуации, в которых он используется, существенно различаются. Так как при этом происходит аварийное завершение программы, этот механизм обычно используется для грубых ошибок, таких как логическая несогласованность в программе; прилежные программисты рассматривают любую аварию кода как доказательство наличия ошибки в программе. В надежной программе "ожидаемые" ошибки, т.е. те, которые возникают в результате неправильного ввода, неверной конфигурации или сбоя ввода-вывода, должны быть корректно обработаны; лучше всего работать с ними с использованием значений `error`.

Рассмотрим функцию `regexp.Compile`, которая компилирует регулярное выражение в эффективную форму для дальнейшего сопоставления. Она возвращает `error`, если функция вызывается с неправильно сформированным шаблоном, но проверка этой ошибки является излишней и обременительной, если вызывающая функция знает, что определенный вызов не может быть неудачным. В таких случаях разумно, чтобы вызывающая функция обработала ошибку с помощью генерации аварийной ситуации, так как ошибка считается невозможной.

Поскольку большинство регулярных выражений представляют собой литералы в исходном тексте программы, пакет `regexp` предоставляет функцию-оболочку `regexp.MustCompile`, выполняющую такую проверку:

```
package regexp

func Compile(expr string) (*Regexp, error) { /* ... */ }

func MustCompile(expr string) *Regexp {
    re, err := Compile(expr)
    if err != nil {
```

```
        panic(err)
    }
    return re
}
```

Эта функция-оболочка позволяет клиентам удобно инициализировать переменные уровня пакета скомпилированным регулярным выражением, как показано ниже:

```
var httpSchemeRE=regexp.MustCompile(`^https?:`) // "http:" или "https:"
```

Конечно, функция `MustCompile` не должна вызываться с недоверенными входными значениями. Префикс `Must` является распространенным соглашением именования для функций такого рода наподобие `template.Must` в разделе 4.6.

Когда программа сталкивается с аварийной ситуацией, все отложенные функции выполняются в порядке, обратном их появлению в исходном тексте, начиная с функции на вершине стека и опускаясь до функции `main`, что демонстрирует приведенная ниже программа:

gopl.io/ch5/defer1

```
func main() {
    f(3)
}

func f(x int) {
    fmt.Printf("f(%d)\n", x+0/x) // Сбой при x == 0
    defer fmt.Printf("defer %d\n", x)
    f(x-1)
}
```

При запуске эта программа выводит на стандартный вывод следующее:

```
f(3)
f(2)
f(1)
defer 1
defer 2
defer 3
```

При вызове `f(0)` возникает аварийная ситуация, приводящая к выполнению трех отложенных вызовов `fmt.Printf`. Затем среда выполнения прекращает выполнение программы, выводя соответствующее аварийное сообщение и дамп стека в стандартный поток ошибок (здесь для ясности вывод упрощен):

```
panic: runtime error: integer divide by zero
main.f(0)
    src/gopl.io/ch5/defer1/defer.go:14
main.f(1)
    src/gopl.io/ch5/defer1/defer.go:16
main.f(2)
    src/gopl.io/ch5/defer1/defer.go:16
main.f(3)
```

```
    src/gopl.io/ch5/defer1/defer.go:16
main.main()
    src/gopl.io/ch5/defer1/defer.go:10
```

Как мы вскоре увидим, функция может восстановиться после аварийной ситуации так, что программа при этом не будет аварийно завершена.

Для диагностических целей пакет `runtime` позволяет программисту вывести дамп стека с помощью того же механизма. Откладывая вызов `printStack` в функции `main`,

gopl.io/ch5/defer2

```
func main() {
    defer printStack()
    f(3)
}

func printStack() {
    var buf [4096]byte
    n := runtime.Stack(buf[:], false)
    os.Stdout.Write(buf[:n])
}
```

мы обеспечиваем вывод дополнительного текста в стандартный поток вывода (вновь упрощенный здесь для ясности):

```
goroutine 1 [running]:
main.printStack()
    src/gopl.io/ch5/defer2/defer.go:20
main.f(0)
    src/gopl.io/ch5/defer2/defer.go:27
main.f(1)
    src/gopl.io/ch5/defer2/defer.go:29
main.f(2)
    src/gopl.io/ch5/defer2/defer.go:29
main.f(3)
    src/gopl.io/ch5/defer2/defer.go:29
main.main()
    src/gopl.io/ch5/defer2/defer.go:15
```

Читатели, знакомые с исключениями в других языках программирования, могут быть удивлены тем, что функция `runtime.Stack` позволяет вывести информацию о функциях, которые кажутся уже "развернутыми". Но механизм аварийной ситуации Go запускает отсроченные функции *до* разворачивания стека.

5.10. Восстановление

Все, что описано выше, — это обычно правильная реакция на аварийную ситуацию, но не всегда. Может возникнуть ситуация, когда возможно некоторое восстановление после ошибки, или по крайней мере возможна "уборка беспорядка" перед

выходом из программы. Например, веб-сервер, который сталкивается с неожиданной проблемой, может аккуратно закрыть подключения, а не оставлять клиентов "зависшими", а во время разработки сообщать клиентам об ошибке.

Если в отложенной функции вызывается встроенная функция `recover` и функция, содержащая инструкцию `defer`, сбоит, `recover` завершает текущее аварийное состояние и возвращает соответствующее значение аварийной ситуации. Функция, которая столкнулась с аварийной ситуацией, продолжается с того места, где она была прервана, и выход из нее осуществляется в штатном режиме. Если `recover` вызывается в любой другой момент, она ничего не делает и возвращает значение `nil`.

Для иллюстрации рассмотрим разработку синтаксического анализатора языка. Даже если нам кажется, что он отлично работает, в нем по-прежнему могут быть ошибки, в особенности если учесть сложность его работы. Мы могли бы предпочесть, чтобы вместо аварийного завершения работы анализатор превращал аварийные ситуации в обычные ошибки синтаксического анализа, возможно, с дополнительным сообщением, предлагающим пользователям сообщить об этой ошибке разработчикам:

```
func Parse(input string) (s *Syntax, err error) {
    defer func() {
        if p := recover(); p != nil {
            err = fmt.Errorf("внутренняя ошибка: %v", p)
        }
    }()
    // ... Синтаксический анализатор ...
}
```

Отложенная функция в `Parse` выполняет восстановление после аварийной ситуации, используя значение аварийной ситуации для создания сообщения об ошибке; более мощная версия может включать весь стек вызовов, полученный с помощью `runtime.Stack`. Затем отложенная функция выполняет присваивание результата переменной `err`, которая возвращается вызывающей функции.

Огульное восстановление после аварийной ситуации представляется сомнительным, потому что состояния переменных пакета после аварии редко точно определены или документированы. Возможно, было неполным критическое обновление структуры данных, файл (или сетевое подключение) был открыт, но не закрылся или блокировка была захвачена, но не освобождена. Кроме того, замена аварийного завершения, например, строкой в файле журнала может привести к тому, что ошибка останется попросту незамеченной.

Восстановление из аварийного состояния в том же пакете может помочь упростить обработку сложных или неожиданных ошибок, но, как правило, не следует пытаться осуществлять восстановление при аварийной ситуации в другом пакете. Общедоступные API должны сообщать о неудачах с помощью `error`. Аналогично вы не должны осуществлять восстановление в случае аварийной ситуации, которая может пройти через не поддерживаемую вами функцию (например, в случае применения функций обратного вызова), так как при этом вы не можете оценить безопасность этого действия.

Например, пакет `net/http` предоставляет веб-сервер, который диспетчеризует входящие запросы, передавая их функциям-обработчикам, предоставляемым пользователями. Вместо того чтобы позволить аварийной ситуации в одном из этих обработчиков завершить весь процесс, сервер вызывает функцию `recover`, выводит трассировку стека и продолжает обслуживание. На практике это удобно, но при таком подходе имеется риск утечки ресурсов или возникновения ситуации, когда аварийный обработчик остается в неопределенном состоянии, что позже может привести к другим проблемам.

По всем указанным выше причинам наиболее безопасным является выборочное восстановление (если оно вообще используется). Другими словами, восстановление может выполняться только в (крайне редких!) аварийных ситуациях, предназначенных для восстановления. Это предназначение может быть закодировано с использованием специального неэкспортируемого типа для значения аварийной ситуации и проверки, имеет ли возвращенное `recover` значение этот тип. (Один из способов, как это сделать, представлен в следующем примере). Если это так, то об аварийной ситуации мы сообщаем как об обычной ошибке; если нет, то мы вызываем `panic` с тем же значением для восстановления аварийного состояния.

Приведенный ниже пример является вариацией программы `title`, которая сообщает об ошибке, если HTML-документ содержит несколько элементов `<title>`. В этом случае программа прерывает рекурсию, вызывая функцию `panic` со значением специального типа `bailout`.

gopl.io/ch5/title3

```
// soleTitle возвращает текст первого непустого элемента title
// в документе и ошибку, если их количество не равно одному.

func soleTitle(doc *html.Node) (title string, err error) {
    type bailout struct{}
    defer func() {
        switch p := recover(); p {
        case nil:
        // no panic
        case bailout{}:
            // "Ожидаемая" аварийная ситуация
            err = fmt.Errorf("несколько элементов title")
        default:
            panic(p) // Неожиданная аварийная ситуация;
                     // восстанавливаем ее
        }
    }()
    // Прекращаем рекурсию при нескольких непустых элементах title.
    forEachNode(doc, func(n *html.Node) {
        if n.Type == html.ElementNode && n.Data == "title" &&
            n.FirstChild != nil {
            if title != "" {
                panic(bailout{}) // Несколько элементов title
```

```
            }
            title = n.FirstChild.Data
        }
    }, nil)
    if title == "" {
        return "", fmt.Errorf("нет элемента title")
    }
    return title, nil
}
```

Отложенная функция-обработчик вызывает `recover`, проверяет значение аварийной ситуации и сообщает об обычной ошибке, если это значение — `bailout{}`. Все прочие значения, не являющиеся `nil`, указывают на неожиданную аварийную ситуацию, и в этом случае обработчик вызывает функцию `panic` с данным значением. Тем самым действие `recover` отменяется и восстанавливается исходное аварийное состояние. (В этом примере несколько нарушены наши советы не использовать аварийную ситуацию для "ожидаемых" ошибок, но это сделано для того, чтобы компактно проиллюстрировать данный механизм.)

Из некоторых ситуаций восстановление невозможно. Например, исчерпание памяти приводит к завершению программы с фатальной ошибкой.

Упражнение 5.19. Воспользуйтесь функциями `panic` и `recover` для написания функции, которая не содержит инструкцию `return`, но возвращает ненулевое значение.

6

Методы

С начала 1990-х годов объектно-ориентированное программирование (ООП) стало доминирующей парадигмой программирования в промышленности и образовании, и почти все широко используемые языки, разработанные с того времени, включают его поддержку. Не является исключением и Go.

Хотя общепринятого определения объектно-ориентированного программирования нет, для наших целей определим, что *объект* представляет собой просто значение или переменную, которая имеет методы, а *метод* — это функция, связанная с определенным типом. Объектно-ориентированная программа — это программа, которая использует методы для выражения свойств и операций каждой структуры данных так, что клиентам не требуется прямой доступ к представлению объекта.

В предыдущих главах мы регулярно использовали методы из стандартной библиотеки, такие как метод `Seconds` типа `time.Duration`:

```
const day = 24 * time.Hour
fmt.Println(day.Seconds()) // "86400"
```

Мы также определяли собственный метод в разделе 2.5 — метод `String` типа `Celsius`:

```
func (c Celsius) String() string { return fmt.Sprintf("%g°C", c) }
```

В этой главе, первой из двух, посвященных объектно-ориентированному программированию, мы покажем, как эффективно определять и использовать методы. Мы также рассмотрим два ключевых принципа объектно-ориентированного программирования — *инкапсуляцию* и *композицию*.

6.1. Объявления методов

Метод объявляется с помощью вариации объявления обычных функций, в котором перед именем функции появляется дополнительный параметр. Этот параметр присоединяет функцию к типу этого параметра.

Давайте напишем наш первый метод в простом пакете для геометрии на плоскости:

```
gopl.io/ch6/geometry
package geometry
```

```
import "math"

type Point struct{ X, Y float64 }

// Традиционная функция
func Distance(p, q Point) float64 {
    return math.Hypot(q.X-p.X, q.Y-p.Y)
}

// То же, но как метод типа Point
func (p Point) Distance(q Point) float64 {
    return math.Hypot(q.X-p.X, q.Y-p.Y)
}
```

Дополнительный параметр p называется *получателем* метода приемника; это название унаследовано от ранних объектно-ориентированных языков, которые описывали вызов метода как "отправку сообщения объекту".

В Go не используется специальное имя, такое как `this` или `self`, для получателя; мы выбираем имя для получателя так же, как для любого другого параметра. Поскольку имя получателя будет использоваться часто, лучше выбрать что-то короткое и согласованно используемое во всех методах. Распространенным выбором является первая буква имени типа, как выше использовано имя p для `Point`.

В вызове метода аргумент получателя находится перед именем метода. Это напоминает объявление, в котором параметр получателя находится перед именем метода:

```
p := Point{1, 2}
q := Point{4, 6}
fmt.Println(Distance(p, q)) // "5", вызов функции
fmt.Println(p.Distance(q))  // "5", вызов метода
```

Нет никакого конфликта между двумя объявлениями функций по имени `Distance`, приведенными выше. Первое объявляет функцию уровня пакета под названием `geometry.Distance`. Второе объявляет метод типа `Point`, поэтому его имя — `Point.Distance`.

Выражение `P.Distance` называется *селектором* (selector), потому что оно выбирает подходящий метод `Distance` для получателя p типа `Point`. Селекторы используются также для выбора полей структурных типов, как в выражении `p.X`. Поскольку методы и поля находятся в одном и том же пространстве имен, объявление в нем метода X для структуры типа `Point` приведет к неоднозначности, и компилятор его отвергнет.

Поскольку каждый тип имеет собственное пространство имен для методов, мы можем использовать имя `Distance` для других методов, лишь бы они принадлежали различным типам. Давайте определим тип `Path`, который представляет собой последовательность отрезков линии, и определим метод `Distance` и для него:

```
// Path — путь из точек, соединенных прямолинейными отрезками.
type Path []Point
```

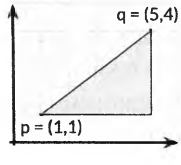
```
// Distance возвращает длину пути.
func (path Path) Distance() float64 {
    sum := 0.0
    for i := range path {
        if i > 0 {
            sum += path[i-1].(path[i])
        }
    }
    return sum
}
```

Path является именованным типом среза, а не структурой, как Point, но мы по-прежнему можем определить для него методы. Разрешая связывать методы с любым типом, Go отличается от многих других объектно-ориентированных языков программирования. Часто бывает удобно определить дополнительное поведение для простых типов, таких как числа, строки, срезы, отображения, а иногда даже функции. Методы могут быть объявлены для любого именованного типа, определенного в том же пакете, лишь бы его базовый тип не являлся ни указателем, ни интерфейсом.

Два рассмотренных метода Distance имеют различные типы. Они не связаны один с другим, хотя Path.Distance использует Point.Distance внутренне для вычисления длины каждого отрезка, соединяющего соседние точки.

Давайте вызовем новый метод для вычисления периметра прямоугольного треугольника:

```
perim := Path{
    {1, 1},
    {5, 1},
    {5, 4},
    {1, 1},
}
fmt.Println(perim.Distance()) // "12"
```



В двух показанных выше вызовах методов Distance компилятор определяет, какая функция должна быть вызвана, основываясь на имени метода и типе получателя. В первом вызове path[i-1] имеет тип Point, поэтому вызывается Point.Distance; во втором perim имеет тип Path, поэтому вызывается метод Path.Distance.

Все методы данного типа должны иметь уникальные имена, но одно и то же имя метода может использоваться разными типами, как метод Distance использован для типов Point и Path. Нет необходимости квалифицировать имена функций (например, PathDistance) для устранения неоднозначности. Здесь мы видим первое преимущество использования методов по сравнению с обычными функциями: имена методов могут быть короче. Преимущество увеличивается для вызовов, выполняемых за пределами пакета, так как они могут использовать более короткое имя *и* опускать имя пакета:

```
import "gopl.io/ch6/geometry"

perim := geometry.Path{{1, 1}, {5, 1}, {5, 4}, {1, 1}}
fmt.Println(geometry.PathDistance(perim)) // "12", автономная функция
fmt.Println(perim.Distance())             // "12", метод geometry.Path
```

6.2. Методы с указателем в роли получателя

Вызов функции создает копию каждого значения аргумента, поэтому, если функции необходимо обновить переменную или если аргумент является настолько большим, что желательно избежать его копирования, следует передавать адрес переменной с помощью указателя. То же самое справедливо и для методов, которым необходимо обновить переменную получателя: их следует присоединять к типу указателя, такому как `*Point`:

```
func (p *Point) ScaleBy(factor float64) {
    p.X *= factor
    p.Y *= factor
}
```

Именем данного метода является is `(*Point).ScaleBy`. Скобки необходимы — без них выражение будет трактоваться как `*(Point.ScaleBy)`.

В реальных программах соглашение диктует, что если какой-либо метод `Point` имеет получатель-указатель, то *все* методы `Point` должны иметь указатель в качестве получателя, даже те, которым это не требуется в обязательном порядке. Мы нарушили это правило для `Point`, чтобы показать вам обе разновидности методов.

Именованные типы (`Point`) и указатели на них (`*Point`) — единственные типы, которые могут появляться в объявлении получателя. Кроме того, чтобы избежать неоднозначности, объявления методов не разрешены для именованных типов, которые сами являются типами указателей:

```
type P *int
func (P) f() {/* ... */} // Ошибка компиляции: неверный тип получателя
```

Метод `(*Point).ScaleBy` может быть вызван с помощью предоставления получателя `*Point`, например, так:

```
r := &Point{1, 2}
r.ScaleBy(2)
fmt.Println(*r) // "{2, 4}"
```

Или так:

```
p := Point{1, 2}
pptr := &p
pptr.ScaleBy(2)
fmt.Println(p) // "{2, 4}"
```

Или вот так:

```
p := Point{1, 2}
(&p).ScaleBy(2)
fmt.Println(p) // "{2, 4}"
```

Но последние два варианта несколько громоздки. К счастью, здесь нам помогает язык. Если получатель `p` является *переменной* типа `Point`, но методу необходим получатель `*Point`, можно использовать сокращенную запись

```
p.ScaleBy(2)
```

при которой компилятор выполнит неявное получение адреса `&p` этой переменной. Это работает только для переменных, включая поля структур наподобие `p.X` и элементы массивов или срезов наподобие `perim[0]`. Мы не можем вызвать метод `*Point` для неадресуемого получателя `Point`, так как нет никакого способа получения адреса временного значения.

```
Point{1, 2}.ScaleBy(2) // Ошибка компиляции: невозможно получить
                       // адрес литерала Point
```

Но мы *можем* вызвать метод `Point` наподобие `Point.Distance` с получателем `*Point`, поскольку есть способ получить значение из адреса: нужно просто загрузить значение, на которое указывает получатель. Компилятор, по сути, вставляет неявный оператор `*`. Два следующих вызова являются эквивалентными:

```
pptr.Distance(q)
(*pptr).Distance(q)
```

Подведем итоги, так как недостаточное понимание в этой области зачастую приводит к путанице. В каждом корректном выражении вызова метода истинным является только одно из трех перечисленных утверждений.

Либо аргумент получателя имеет тот же тип, что и параметр получателя, например оба имеют тип `T` или оба имеют тип `*T`:

```
Point{1, 2}.Distance(q) // Point
pptr.ScaleBy(2)         // *Point
```

Либо аргумент получателя является *переменной* типа `T`, а параметр получателя имеет тип `*T`. Компилятор неявно получает адрес переменной:

```
p.ScaleBy(2)            // Неявное (&p)
```

Либо аргумент получателя имеет тип `*T`, а параметр получателя имеет тип `T`. Компилятор неявно разыменовывает получателя, другими словами, загружает соответствующее значение:

```
pptr.Distance(q)        // Неявное (*pptr)
```

Если все методы именованного типа `T` имеют тип получателя `T` (не `*T`), то копирование экземпляров этого типа безопасно; вызов любого из его методов обязательно делает копию. Например, значения `time.Duration` свободно копируются, в том числе в качестве аргументов функций. Но если какой-либо метод имеет в качестве получателя указатель, следует избегать копирования экземпляров `T`, потому

что это может нарушать внутренние инварианты. Например, копирование экземпляра `bytes.Buffer` может привести к тому, что оригинал и копия будут псевдонимами (раздел 2.3.2) одного и того же базового массива байтов, и последующие вызовы методов будут иметь непредсказуемые результаты.

6.2.1. Значение `nil` является корректным получателем

Так же, как некоторые функции допускают нулевые указатели в качестве аргументов, так и некоторые методы допускают нулевые указатели в качестве их получателя, особенно если `nil` является полноценным нулевым значением типа, как в случае отображений и срезов. В приведенном далее простом связанном списке целых чисел значение `nil` представляет пустой список:

```
// IntList представляет собой связанный список целых чисел.
// Значение *IntList, равное nil, представляет пустой список.
type IntList struct {
    Value int
    Tail *IntList
}

// Sum возвращает сумму элементов списка.
func (list *IntList) Sum() int {
    if list == nil {
        return 0
    }
    return list.Value + list.Tail.Sum()
}
```

При определении типа, методы которого допускают нулевое значение получателя, желательно явно указать это в документирующих комментариях, как это сделано выше.

Вот часть определения типа `Values` из пакета `net/url`:

net/url

```
package url

// Values отображает строковый ключ на список значений.
type Values map[string][]string

// Get возвращает первое значение, связанное с данным ключом,
// или "", если такового нет.
func (v Values) Get(key string) string {
    if vs := v[key]; len(vs) > 0 {
        return vs[0]
    }
    return ""
}
```

```
// Add добавляет значение к ключу. Добавление выполняется к любым
// существующим значениям, связанным с ключом.
func (v Values) Add(key, value string) {
    v[key] = append(v[key], value)
}
```

Оно показывает, что представлением данного типа является отображение, а также предоставляет методы для доступа к отображению, значениями которого являются срезы строк — это *мультиотображение*. Клиенты могут использовать его встроенные операторы (`make`, литералы срезов, `m[key]` и т.д.) или его методы, или и то, и другое вместе, что им больше нравится:

gopl.io/ch6/urlvalues

```
m := url.Values{"lang": {"en"}} // Непосредственное создание
m.Add("item", "1")
m.Add("item", "2")

fmt.Println(m.Get("lang")) // "en"
fmt.Println(m.Get("q"))    // ""
fmt.Println(m.Get("item")) // "1" (первое значение)
fmt.Println(m["item"])     // "[1 2]" (непосредственное обращение)

m = nil
fmt.Println(m.Get("item")) // ""
m.Add("item", "3")         // Аварийная ситуация: присваивание
                           // записи в пустом отображении
```

В последнем вызове `Get` нулевой получатель ведет себя, как пустое отображение. Мы могли бы эквивалентно записать вызов как `Values(nil).Get("Item")`, но выражение `nil.Get("item")` не будет компилироваться, потому что тип `nil` не определен. Последний же вызов `Add` приводит к аварийной ситуации, поскольку он пытается обновить нулевое отображение.

Поскольку `url.Values` имеет тип отображения, а отображение обращается к своим парам "ключ–значение" косвенно, любые обновления и удаления, которые делают вызовы `url.Values.Add` с элементами отображений, видны вызывающей функции. Однако, как и в случае обычных функций, любые изменения, которые метод делает с самой ссылкой, такие как установка ее значения равным `nil` или ее перенаправление на другое отображение, не будет видимо вызывающей функции.

6.3. Создание типов путем встраивания структур

Рассмотрим тип `ColoredPoint`:

gopl.io/ch6/coloredpoint

```
import "image/color"
```

```
type Point struct{ X, Y float64 }
type ColoredPoint struct {
    Point
    Color color.RGBA
}
```

Мы могли бы определить `ColoredPoint` как структуру из трех полей, но вместо этого *встраиваем* `Point` для предоставления полей `X` и `Y`. Как мы видели в разделе 4.4.3, встраивание позволяет нам использовать синтаксические сокращения для определения структуры `ColoredPoint`, которая содержит все поля `Point` плюс еще некоторые. При желании мы можем выбрать поля `ColoredPoint`, которые были предоставлены встроенной структурой `Point` без упоминания имени `Point`:

```
var cp ColoredPoint
cp.X = 1
fmt.Println(cp.Point.X) // "1"
cp.Point.Y = 2
fmt.Println(cp.Y)       // "2"
```

Аналогичный механизм применим и к *методам* `Point`. Мы можем вызывать методы встроенного поля `Point` с использованием получателя типа `ColoredPoint`, несмотря на то что `ColoredPoint` не имеет объявленных методов:

```
red  := color.RGBA{255, 0, 0, 255}
blue := color.RGBA{0, 0, 255, 255}
var p = ColoredPoint{Point{1, 1}, red}
var q = ColoredPoint{Point{5, 4}, blue}
fmt.Println(p.Distance(q.Point)) // "5"
p.ScaleBy(2)
q.ScaleBy(2)
fmt.Println(p.Distance(q.Point)) // "10"
```

Методы `Point` были *повышены* до методов `ColoredPoint`. Таким образом, встраивание допускает наличие сложных типов со многими методами, при этом указанные типы создаются путем *композиции* нескольких полей, каждое из которых предоставляет несколько методов.

Читатели, знакомые с объектно-ориентированными языками, основанными на классах, могут соблазниться провести параллели и рассматривать `Point` как базовый класс, а `ColoredPoint` — как подкласс (или производный класс) или интерпретировать связь между этими типами, как если бы `ColoredPoint` "являлся" `Point`. Но это было бы ошибкой. Обратите внимание на показанные выше вызовы `Distance`. Метод `Distance` имеет параметр типа `Point`, но `q` не является `Point`, так что хотя `q` и имеет встроенные поля этого типа, мы должны явно их указать. Попытка передачи `q` приведет к сообщению об ошибке:

```
p.Distance(q) // Ошибка компиляции: нельзя использовать
              // q (ColoredPoint) как Point
```

`ColoredPoint` не является `Point`, но "содержит" `Point` и имеет два дополнительных метода (`Distance` и `ScaleBy`), повышенных из `Point`. Если вы предпочитаете думать в терминах реализации, встроенное поле указывает компилятору на необходимость генерации дополнительных "методов-оберток", которые делегируют вызов объявленным методам что-то вроде такого кода:

```
func (p ColoredPoint) Distance(q Point) float64 {
    return p.Point.Distance(q)
}

func (p *ColoredPoint) ScaleBy(factor float64) {
    p.Point.ScaleBy(factor)
}
```

Когда `Point.Distance` вызывается первым из этих методов-оберток, значением его получателя является `p.Point`, а не `p`, и при этом нет никакого способа, которым метод мог бы обратиться к структуре `ColoredPoint`, в которую встроена структура `Point`.

Типом анонимного поля может быть указатель на именованный тип; в этом случае поля и методы косвенно повышаются из указываемого объекта. Добавление еще одного уровня косвенности позволяет нам совместно использовать общие структуры и динамически изменять взаимоотношения между объектами. Приведенное ниже объявление `ColoredPoint` встраивает `*Point`:

```
type ColoredPoint struct {
    *Point
    Color color.RGBA
}

p := ColoredPoint{&Point{1, 1}, red}
q := ColoredPoint{&Point{5, 4}, blue}
fmt.Println(p.Distance(*q.Point)) // "5"
q.Point = p.Point                 // p и q разделяют одну и ту же Point
p.ScaleBy(2)
fmt.Println(*p.Point, *q.Point)   // "{2 2} {2 2}"
```

Структурный тип может иметь более одного анонимного поля. Если мы объявим `ColoredPoint` как

```
type ColoredPoint struct {
    Point
    color.RGBA
}
```

то значение этого типа будет иметь все методы типа `Point`, все методы `RGBA` и любые дополнительные методы, непосредственно объявленные для `ColoredPoint`. Когда компилятор разрешает селектор как `p.ScaleBy` для вызова метода, сначала он ищет непосредственно объявленный метод с именем `ScaleBy`, затем — метод, однократно повышенный из встроенных полей `ColoredPoint`, после этого — дважды повы-

шенный метод из встроенных внутри Point и RGBA, и т.д. Компилятор сообщает об ошибке, если селектор является неоднозначным из-за того, что имеется два или более методов с одним именем с одинаковым рангом повышения.

Методы могут быть объявлены только для именованных типов (наподобие Point) и указателей на них (*Point), но благодаря встраиванию *неименованные* структурные типы также могут иметь методы (иногда это оказывается полезным).

Вот неплохая иллюстрация к сказанному. В приведенном примере показана часть простого кеша, реализованного с помощью двух переменных уровня пакета, мьютекса (раздел 9.2) и отображения, которое он защищает:

```
var (
    mu sync.Mutex     // Защищает отображение
    mapping = make(map[string]string)
)

func Lookup(key string) string {
    mu.Lock()
    v := mapping[key]
    mu.Unlock()
    return v
}
```

Приведенная ниже версия функционально эквивалентна, но группирует эти две связанные переменные вместе в одну переменную уровня пакета cache:

```
var cache = struct {
    sync.Mutex
    mapping map[string]string
} {
    mapping: make(map[string]string),
}

func Lookup(key string) string {
    cache.Lock()
    v := cache.mapping[key]
    cache.Unlock()
    return v
}
```

Новая переменная дает связанным с кешем переменным более выразительные имена, а поскольку в нее встроено поле sync.Mutex, его методы Lock и Unlock повышаются до неименованного структурного типа, позволяя нам блокировать cache с помощью самоочевидного синтаксиса.

6.4. Значения-методы и выражения-методы

Обычно мы выбираем и вызываем метод в одном выражении, как, например, в p.Distance(), но эти две операции можно разделить. Селектор p.Distance дает

нам *значение-метод* (method value), функцию, которая связывает метод (`Point.Distance`) со значением конкретного получателя `p`. Эта функция может быть вызвана без указания значения получателя; ей нужны только аргументы, не являющиеся получателем.

```
p := Point{1, 2}
q := Point{4, 6}

distanceFromP := p.Distance        // Значение-метод
fmt.Println(distanceFromP(q))      // "5"
var origin Point                   // {0, 0}
fmt.Println(distanceFromP(origin)) // "2.23606797749979", √5

scaleP := p.ScaleBy                // Значение-метод
scaleP(2)                          // p становится (2, 4)
scaleP(3)                          // затем (6, 12)
scaleP(10)                         // затем (60, 120)
```

Значения-методы полезны, когда API пакета требует значение-функцию, а для клиента желаемым поведением для этой функции является вызов метода для конкретного получателя. Например, функция `time.AfterFunc` вызывает значение-функцию после заданной задержки. Приведенная программа использует ее для запуска ракеты `r` через 10 секунд:

```
type Rocket struct { /* ... */ }
func (r *Rocket) Launch() { /* ... */ }

r := new(Rocket)
time.AfterFunc(10 * time.Second, func() { r.Launch() })
```

Синтаксис с использованием значения-метода оказывается более коротким:

```
time.AfterFunc(10 * time.Second, r.Launch)
```

Со значениями-методами тесно связаны *выражения-методы*. При вызове метода, в противоположность обычной функции, мы должны указать получателя с помощью синтаксиса селектора. Выражение-метод, записываемое как `T.f` или `(*T).f`, где `T` — тип, дает значение-функцию с обычным первым параметром, представляющим собой получатель, так что его можно вызывать, как обычную функцию:

```
p := Point{1, 2}
q := Point{4, 6}

distance := Point.Distance   // Выражение-метод
fmt.Println(distance(p, q))  // "5"
fmt.Printf("%T\n", distance) // "func(Point, Point) float64"

scale := (*Point).ScaleBy
scale(&p, 2)
fmt.Println(p)               // "{2 4}"
fmt.Printf("%T\n", scale)    // "func(*Point, float64)"
```

Выражения-методы могут быть полезны, когда требуется значение для представления выбора среди нескольких методов, принадлежащих одному типу, такое, чтобы выбранный метод можно было вызвать со многими различными получателями. В следующем примере переменная op представляет метод сложения либо вычитания типа Point, и Path.TranslateBy называет его для каждой точки в пути Path:

```
type Point struct{ X, Y float64 }

func (p Point) Add(q Point) Point { return Point{p.X+q.X, p.Y+q.Y} }
func (p Point) Sub(q Point) Point { return Point{p.X-q.X, p.Y-q.Y} }

type Path []Point

func (path Path) TranslateBy(offset Point, add bool) {
    var op func(p, q Point) Point
    if add {
        op = Point.Add
    } else {
        op = Point.Sub
    }
    for i := range path {
        // Вызов либо path[i].Add(offset), либо path[i].Sub(offset).
        path[i] = op(path[i], offset)
    }
}
```

6.5. Пример: тип битового вектора

Множества в Go обычно реализуются как map[T]bool, где T является типом элемента. Множество, представленное отображением, очень гибкое, но для некоторых задач специализированное представление может его превзойти. Например, в таких областях, как анализ потока данных, где элементы множества представляют собой небольшие неотрицательные целые числа, множества имеют много элементов, а распространенными операциями являются объединение и пересечение множеств, идеальным решением оказывается *битовый вектор*.

Битовый вектор использует срез беззнаковых целочисленных значений, или "слов", каждый бит которых представляет возможный элемент множества. Множество содержит *i*, если *i*-й бит установлен. Приведенная далее программа демонстрирует простой тип битового вектора с тремя методами:

```
gopl.io/ch6/intset
// IntSet представляет собой множество небольших неотрицательных
// целых чисел. Нулевое значение представляет пустое множество.

type IntSet struct {
    words []uint64
```

```
}

// Has указывает, содержит ли множество неотрицательное значение x.
func (s *IntSet) Has(x int) bool {
    word, bit := x/64, uint(x%64)
    return word < len(s.words) && s.words[word]&(1<<bit) != 0
}

// Add добавляет неотрицательное значение x в множество.
func (s *IntSet) Add(x int) {
    word, bit := x/64, uint(x%64)
    for word >= len(s.words) {
        s.words = append(s.words, 0)
    }
    s.words[word] |= 1 << bit
}

// UnionWith делает множество s равным объединению множеств s и t.
func (s *IntSet) UnionWith(t *IntSet) {
    for i, tword := range t.words {
        if i < len(s.words) {
            s.words[i] |= tword
        } else {
            s.words = append(s.words, tword)
        }
    }
}
```

Поскольку каждое слово имеет 64 бита, чтобы найти бит для значения x, мы используем частное x/64 в качестве индекса слова, а остаток x%64 — как индекс бита внутри этого слова. Операция UnionWith использует оператор побитового ИЛИ (|) для вычисления объединения 64 элементов за один раз. (Мы вернемся к выбору 64-битовых слов в упражнении 6.5.)

В данной реализации не хватает многих функций, которые хотелось бы иметь. Некоторые из них оставлены в качестве упражнений читателям, но без одной из них очень трудно обойтись: это вывод множества IntSet в виде строки. Давайте добавим к этому типу метод String, как мы уже делали для типа Celsius в разделе 2.5:

```
// String возвращает множество как строку вида "{1 2 3}".
func (s *IntSet) String() string {
    var buf bytes.Buffer
    buf.WriteByte('{')
    for i, word := range s.words {
        if word == 0 {
            continue
        }
        for j := 0; j < 64; j++ {
            if word&(1<<uint(j)) != 0 {
```

```
                if buf.Len() > len("{") {
                    buf.WriteByte(' ')
                }
                fmt.Fprintf(&buf, "%d", 64*i+j)
            }
        }
    }
    buf.WriteByte('}')
    return buf.String()
}
```

Обратите внимание на схожесть метода `String` в приведенном выше исходном тексте с `intsToString` в разделе 3.5.4; `bytes.Buffer` часто используется таким образом в методах `String`. Пакет `fmt` рассматривает типы с методом `String` специальным образом, так, чтобы значения сложных типов можно было выводить в удобочитаемом виде. Вместо вывода неформатированного представления значения (в данном случае — структуры) `fmt` вызывает метод `String`. Этот механизм опирается на интерфейсы и утверждения о типах, с которыми мы встретимся в главе 7.

Теперь можно продемонстрировать `IntSet` в действии:

```
var x, y IntSet
x.Add(1)
x.Add(144)
x.Add(9)
fmt.Println(x.String())           // "{1 9 144}"

y.Add(9)
y.Add(42)
fmt.Println(y.String())           // "{9 42}"

x.UnionWith(&y)
fmt.Println(x.String())           // "{1 9 42 144}"

fmt.Println(x.Has(9), x.Has(123)) // "true false"
```

Небольшое предостережение: мы объявили `String` и `Has` как методы типа указателя `*IntSet` не по необходимости, а для обеспечения согласованности с двумя другими методами, которым в качестве получателя нужен указатель, поскольку в этих методах выполняется присваивание `s.words`. Следовательно, *значение* `IntSet` не имеет метода `String`, что иногда приводит к таким сюрпризам, как этот:

```
fmt.Println(&x)          // "{1 9 42 144}"
fmt.Println(x.String())  // "{1 9 42 144}"
fmt.Println(x)           // "{[4398046511618 0 65536]}"
```

В первом случае мы выводим указатель `*IntSet`, у которого есть метод `String`. Во втором случае мы вызываем `String()` для переменной `IntSet`; компилятор вставляет неявную операцию &, давая нам указатель, который имеет метод `String`. Но в третьем случае, поскольку значение `IntSet` не имеет метода `String`, `fmt.`

`Println` выводит представление структуры. Важно не забывать оператор `&`. Сделать `String` методом `IntSet`, а не `*IntSet`, может быть хорошей идеей, но это зависит от конкретных обстоятельств.

Упражнение 6.1. Реализуйте следующие дополнительные методы:

```
func (*IntSet) Len() int      // Возвращает количество элементов
func (*IntSet) Remove(x int)  // Удаляет x из множества
func (*IntSet) Clear()        // Удаляет все элементы множества
func (*IntSet) Copy() *IntSet // Возвращает копию множества
```

Упражнение 6.2. Определите вариативный метод `(*IntSet).AddAll(...int)`, который позволяет добавлять список значений, например `s.AddAll(1,2,3)`.

Упражнение 6.3. `(*IntSet).UnionWith` вычисляет объединение двух множеств с помощью оператора `|`, побитового оператора ИЛИ. Реализуйте методы `IntersectWith`, `DifferenceWith` и `SymmetricDifference` для соответствующих операций над множествами. (Симметричная разность двух множеств содержит элементы, имеющиеся в одном из множеств, но не в обоих одновременно.)

Упражнение 6.4. Добавьте метод `Elems`, который возвращает срез, содержащий элементы множества и годящийся для итерирования с использованием цикла по диапазону `range`.

Упражнение 6.5. Типом каждого слова, используемого в `IntSet`, является `uint64`, но 64-разрядная арифметика может быть неэффективной на 32-разрядных платформах. Измените программу так, чтобы она использовала тип `uint`, который представляет собой наиболее эффективный беззнаковый целочисленный тип для данной платформы. Вместо деления на 64 определите константу, в которой хранится эффективный размер `uint` в битах, 32 или 64. Для этого можно воспользоваться, возможно, слишком умным выражением `32<<(^uint(0)>>63)`.

6.6. Инкапсуляция

Переменная (или метод объекта) называется *инкапсулированной*, если она недоступна клиенту этого объекта. Инкапсуляция, иногда именуемая *сокрытием информации*, является ключевым аспектом объектно-ориентированного программирования.

Go имеет только один механизм для управления видимостью имен: идентификаторы, начинающиеся с прописной буквы, экспортируются из пакета, в котором они определены, а начинающиеся со строчной буквы — нет. Такой же механизм, как и ограничивающий доступ к членам пакета, ограничивает доступ и к полям структуры или методам типа. Как следствие, для инкапсуляции объекта мы должны сделать его структурой.

По этой причине тип `IntSet` из предыдущего раздела был объявлен как структурный, несмотря на то что в нем имеется только одно поле:

```
type IntSet struct {
    words []uint64
}
```

Мы могли бы вместо этого определить `IntSet` как тип среза, как показано ниже, хотя, конечно, должны были бы при этом заменить в его методах каждое вхождение `s.words` на `*s`:

```
type IntSet []uint64
```

Хотя эта версия `IntSet` была бы, по сути, эквивалентна имеющейся, она позволяла бы клиентам из других пакетов читать и модифицировать срез непосредственно. Иначе говоря, в то время как выражение `*s` может быть использовано в любом пакете, `s.words` может появиться только в пакете, в котором определен `IntSet`.

Еще одним следствием механизма экспорта, основанного на именах, является то, что единицей инкапсуляции является пакет, а не тип, как во многих других языках программирования. Поля структурного типа являются видимыми для всего кода в том же пакете. Находится ли этот код в функции или методе, не имеет никакого значения.

Инкапсуляция имеет три преимущества. Во-первых, поскольку клиенты не могут изменять переменные объекта непосредственно, необходимо изучать меньше инструкций, чтобы понять, какими могут быть возможные значения этих переменных.

Во-вторых, сокрытие деталей реализации устраняет зависимость клиентов от сущностей, которые могут изменяться, что дает проектировщику большую свободу в развитии реализации без нарушения совместимости API.

В качестве примера рассмотрим тип `bytes.Buffer`. Он часто используется для накопления очень коротких строк, так что выгодной оптимизацией является резервирование некоторого дополнительного пространства в объекте, чтобы избежать излишнего перераспределения памяти в этом распространенном случае. Поскольку `Buffer` представляет собой структурный тип, это пространство принимает вид дополнительного поля `[64]byte` с именем, начинающимся со строчной буквы. После добавления этого поля в силу его неэкспортируемости клиенты `Buffer` вне пакета `bytes` ничего не знают о каких-либо изменениях, за исключением повышения производительности. `Buffer` и его метод `Grow` приведены ниже (немного упрощены для ясности):

```
type Buffer struct {
    buf []byte
    initial [64]byte
    /* ... */
}

// Grow увеличивает при необходимости емкость буфера,
// чтобы гарантировать наличие места для еще n байтов. [...]
func (b *Buffer) Grow(n int) {
    if b.buf == nil {
        b.buf = b.initial[:0] // Изначально используется
    }                         // предварительно выделенная память.
    if len(b.buf)+n > cap(b.buf) {
        buf := make([]byte, b.Len(), 2*cap(b.buf) + n)
        copy(buf, b.buf)
        b.buf = buf
    }
}
```

Третье, и во многих случаях наиболее важное, преимущество инкапсуляции состоит в том, что она не позволяет клиентам произвольным образом устанавливать значения переменных объекта. Поскольку переменные объекта могут устанавливаться только функциями из одного пакета, автор этого пакета может гарантировать, что все функции поддерживают внутренние инварианты объектов. Например, показанный ниже тип `Counter` позволяет клиентам выполнять приращение счетчика или сбрасывать его значение до нуля, но не устанавливать его равным некоторому произвольному значению:

```
type Counter struct { n int }
func (c *Counter) N() int     { return c.n }
func (c *Counter) Increment() { c.n++ }
func (c *Counter) Reset()     { c.n = 0 }
```

Функции, которые просто получают доступ ко внутренним значениям типа или изменяют их, такие как методы типа `Logger` из пакета `log`, показанные ниже, называются методами *получения* и *установки* значения (getter и setter). Однако при именовании метода получения значения мы обычно опускаем префикс `Get`. Это предпочтение краткости относится ко всем методам (не только к методам доступа к полям), а также к прочим избыточным префиксам, таким как `Fetch`, `Find` или `Lookup`:

```
package log

type Logger struct {
    flags int
    prefix string
    // ...
}

func (l *Logger) Flags() int
func (l *Logger) SetFlags(flag int)
func (l *Logger) Prefix() string
func (l *Logger) SetPrefix(prefix string)
```

Стиль Go не запрещает экспортировать поля. Конечно, после экспорта поле не может стать неэкспортируемым без внесения несовместимых изменений в API, поэтому первоначальный выбор должен быть преднамеренным. Также должны быть тщательно рассмотрены вопрос о сложности инвариантов, которые должны поддерживаться, вероятность будущих изменений и количество клиентского кода, который будет затронут внесением изменений.

Инкапсуляция желательна не всегда. Открывая представление числа наносекунд как `int64`, `time.Duration` позволяет использовать все обычные арифметические операции и операции сравнения при работе с периодами времени и даже для определения констант этого типа:

```
const day = 24 * time.Hour
fmt.Println(day.Seconds()) // "86400"
```

В качестве другого примера сравним `IntSet` с типом `geometry.Path` в самом начале этой главы. `Path` был определен как тип среза и позволяет своим клиентам создавать экземпляры с использованием синтаксиса литералов срезов, выполнять итерации по его точкам с помощью цикла по диапазону и так далее, тогда как для клиентов `IntSet` эти операции недоступны.

Вот принципиальное отличие: `geometry.Path`, по сути, является последовательностью точек, не больше и не меньше, и добавление в него новых полей не предвидится, так что для пакета `geometry` имеет смысл показать, что `Path` является срезом. В противоположность этому `IntSet` просто случайно представлен в виде среза `[]uint64`. Он мог бы иметь представление с использованием `[]uint` или чего-то совершенно иного для разреженных или очень малых множеств. Или, быть может, определенное преимущество дадут дополнительные возможности, такие как еще одно поле для записи количества элементов множества. По этим причинам имеет смысл сделать `IntSet` непрозрачным.

Из этой главы вы узнали, как связать методы с именованными типами и как вызывать эти методы. Хотя методы имеют решающее значение для объектно-ориентированного программирования, они представляют только половину картины. Чтобы завершить ее, нужны *интерфейсы*. О них речь пойдет в следующей главе.

7

Интерфейсы

Интерфейсные типы выражают обобщения или абстракции поведения других типов. С помощью обобщения интерфейсы позволяют писать более гибкие и адаптируемые функции, не привязанные к деталям одной конкретной реализации.

Некоторое понятие интерфейсов имеется у многих объектно-ориентированных языков, но отличительной особенностью интерфейсов Go является то, что они *удовлетворяются неявно*. Другими словами, нет необходимости объявлять все интерфейсы, которым соответствует данный конкретный тип; достаточно просто наличия необходимых методов. Такой дизайн позволяет создавать новые интерфейсы, которым соответствуют существующие конкретные типы, без изменения существующих типов, что особенно удобно для типов, определенных в пакетах, которые вы не контролируете.

Эту главу мы начнем с рассмотрения основ механики интерфейсных типов и их значений. Попутно мы изучим несколько важных интерфейсов из стандартной библиотеки. Многие программы Go используют стандартные интерфейсы ничуть не меньше, чем собственные. Наконец мы рассмотрим *декларации типов* (раздел 7.10) и *переключатели типов* (раздел 7.13) и увидим, как они обеспечивают обобщенность иного рода.

7.1. Интерфейсы как контракты

Все типы, которые мы рассматривали до сих пор, были *конкретными типами*. Конкретный тип определяет точное представление своих значений и предоставляет встроенные операции этого представления, например арифметические операции для чисел, или индексацию, добавление значений и цикл по диапазону для срезов. Конкретный тип может также предоставлять дополнительное поведение с помощью своих методов. Если у вас есть значение конкретного типа, вы точно знаете, что оно собой *представляет* и что вы можете с ним *сделать*.

В Go имеется еще одна разновидность типов — *тип интерфейса*. Интерфейс является *абстрактным типом*. Он не раскрывает представление или внутреннюю структуру своих значений, или набор основных поддерживаемых ими операций; он показывает только некоторые из их методов. Если у вас есть значение интерфейсного

типа, вы ничего не знаете о том, чем оно *является*; вы знаете только то, что оно может *делать* или, точнее, какое поведение предоставляется его методами.

Во всей этой книге мы использовали две схожие функции для форматирования строк: `fmt.Printf`, которая записывает результаты в стандартный поток вывода (файл), и `fmt.Sprintf`, которая возвращает результат в виде строки `string`. Было бы грустно, если бы самую сложную часть, форматирование результата, пришлось дублировать из-за поверхностных различий в использовании результатов. Благодаря интерфейсам все оказывается не так грустно. Обе функции являются, по сути, "обертками" для третьей функции, `fmt.Fprintf`, которая ничего не знает о том, что будет с результатом, который она вычисляет:

```
package fmt

func Fprintf(w io.Writer,format string,args ...interface{})(int,error)

func Printf(format string, args ...interface{}) (int, error) {
    return Fprintf(os.Stdout, format, args...)
}

func Sprintf(format string, args ...interface{}) string {
    var buf bytes.Buffer
    Fprintf(&buf, format, args...)
    return buf.String()
}
```

Префикс `F` в `Fprintf` обозначает *файл* и указывает, что отформатированный вывод должен записываться в файл, предоставляемый в качестве первого аргумента. В случае `Printf` аргумент, `os.Stdout`, имеет тип `*os.File`. Однако в случае `Sprintf` аргумент не является файлом, хотя внешне напоминает его: `&buf` — это указатель на буфер в памяти, в который можно записать байты.

Первый параметр `Fprintf` вообще не является файлом. Его тип — `io.Writer`, который является типом интерфейса со следующим объявлением:

```
package io
// Writer является интерфейсом, являющимся оболочкой метода Write.
type Writer interface {
    // Write записывает len(p) байтов из p в базовый поток данных.
    // Метод возвращает количество байтов, записанных из p
    // (0 <= n <= len(p)), а любая ошибка вызывает прекращение записи.
    // Write должен вернуть ненулевую ошибку при n < len(p).
    // Write не должен изменять данные среза, даже временно.
    //
    // Реализации не должны сохранять p.
    Write(p []byte) (n int, err error)
}
```

Интерфейс `io.Writer` определяет контракт между `Fprintf` и вызывающими его функциями. С одной стороны, контракт требует, чтобы вызывающая функция предо-

ставляла значение конкретного типа наподобие `*os.File` или `*bytes.Buffer`, который содержит метод с именем `Write` с соответствующими сигнатурой и поведением. С другой стороны, контракт гарантирует, что `Fprintf` будет выполнять свою работу для любого значения, соответствующего интерфейсу `io.Writer`. Функция `Fprintf` не может считать, что выполняет запись в файл или в память, — она может только вызывать `Write`.

Поскольку `fmt.Fprintf` ничего не предполагает о представлении значения и полагается только на поведение, гарантированное контрактом `io.Writer`, мы можем безопасно передать значение любого конкретного типа, который соответствует интерфейсу `io.Writer`, в качестве первого аргумента `fmt.Fprintf`. Эта свобода замены одного типа другим, который соответствует тому же интерфейсу, называется *взаимозаменяемостью* (substitutability) и является отличительной особенностью объектно-ориентированного программирования.

Давайте проверим это с помощью нового типа. Метод `Write` приведенного ниже типа `*ByteCounter` просто подсчитывает количество записанных в него байтов, после чего игнорирует их. (Для соответствия типа `len(p)` типу `*c` в операторе `+=` необходимо выполнить преобразование типа.)

gopl.io/ch7/bytecounter

```
type ByteCounter int
func (c *ByteCounter) Write(p []byte) (int, error) {
    *c += ByteCounter(len(p)) // Преобразование int в ByteCounter
    return len(p), nil
}
```

Поскольку `*ByteCounter` соответствует контракту `io.Writer`, мы можем передать его в `Fprintf`, который выполняет форматирование строки, даже не замечая этого изменения; `ByteCounter` накапливает длину выводимого результата.

```
var c ByteCounter
c.Write([]byte("hello"))
fmt.Println(c)  // "5", = len("hello")
c = 0           // Сброс счетчика
var name = "Dolly"
fmt.Fprintf(&c, "hello, %s", name)
fmt.Println(c)  // "12", = len("hello, Dolly")
```

Помимо `io.Writer`, имеется еще один очень важный для пакета `fmt` интерфейс. Функции `Fprintf` и `Fprintln` предоставляют типам возможность управлять видом их значений при выводе. В разделе 2.5 мы определили метод `String` для типа `Celsius` таким образом, чтобы температура выводилась как `"100°C"` , а в разделе 6.5 мы оснастили методом `String` тип `*IntSet` для визуализации последнего с использованием традиционного обозначения множества как `"{1 2 3}"`. Объявление метода `String` делает тип соответствующим одному из наиболее широко используемых интерфейсов, `fmt.Stringer`:

```
package fmt
// Метод String используется для вывода значений, передаваемых
```

```
// в качестве операндов любому формату, принимающему строку, или
// неформатированному принтеру, такому как Print.
type Stringer interface {
    String() string
}
```

Как пакет `fmt` выясняет, какие значения соответствуют интерфейсу, мы расскажем в разделе 7.10.

Упражнение 7.1. Используя идеи из `ByteCounter`, реализуйте счетчики для слов и строк. Вам пригодится функция `bufio.ScanWords`.

Упражнение 7.2. Напишите функцию `CountingWriter` с приведенной ниже сигнатурой, которая для данного `io.Writer` возвращает новый `Writer`, являющийся оболочкой исходного, и указатель на переменную `int64`, которая в любой момент содержит количество байтов, записанных в новый `Writer`.

```
func CountingWriter(w io.Writer) (io.Writer, *int64)
```

Упражнение 7.3. Напишите метод `String` для типа `*tree` из `gopl.io/ch4/treesort` (раздел 4.4), который показывает последовательность значений в дереве.

7.2. Типы интерфейсов

Тип интерфейса определяет множество методов, которыми должен обладать конкретный тип, чтобы рассматриваться в качестве экземпляра этого интерфейса.

Тип `io.Writer` является одним из наиболее широко используемых интерфейсов, потому что он предоставляет абстракцию всех типов, в которые можно записывать байты, и включает в себя файлы, буфера памяти, сетевые подключения, HTTP-клиенты, архиваторы и т.д. Пакет `io` определяет много других полезных интерфейсов. `Reader` представляет любой тип, из которого можно читать байты, а `Closer` — любое значение, которое можно закрыть, такое как файл или сетевое подключение. (Теперь вы, вероятно, уже уяснили соглашения об именовании многих интерфейсов Go с единственным методом.)

```
package io

type Reader interface {
    Read(p []byte) (n int, err error)
}

type Closer interface {
    Close() error
}
```

Далее мы находим объявления новых интерфейсных типов как комбинации уже имеющихся. Ниже приведены два примера:

```
type ReadWriter interface {
    Reader
```

```
    Writer
}

type ReadWriteCloser interface {
    Reader
    Writer
    Closer
}
```

Использованный выше синтаксис, напоминающий встраивание структур, позволяет использовать имя другого интерфейса как собирательное для записи всех его методов. Это действие называется внедрением интерфейса. Мы могли бы написать `io.ReadWriter` и без внедрения, хотя и менее кратко:

```
type ReadWriter interface {
    Read(p []byte) (n int, err error)
    Write(p []byte) (n int, err error)
}
```

Или с применением комбинации двух стилей:

```
type ReadWriter interface {
    Read(p []byte) (n int, err error)
    Writer
}
```

Все три приведенные объявления дают один и тот же результат. Порядок, в котором появляются методы, значения не имеет. Важно только множество методов.

Упражнение 7.4. Функция `strings.NewReader` возвращает значение, соответствующее интерфейсу `io.Reader` (и другим), путем чтения из своего аргумента, который представляет собой строку. Реализуйте простую версию `NewReader` и используйте ее для создания синтаксического анализатора HTML (раздел 5.2), принимающего входные данные из строки.

Упражнение 7.5. Функция `LimitReader` из пакета `io` принимает переменную `r` типа `io.Reader` и количество байтов `n` и возвращает другой объект `Reader`, который читает из `r`, но после чтения `n` сообщает о достижении конца файла. Реализуйте его.

```
func LimitReader(r io.Reader, n int64) io.Reader
```

7.3. Соответствие интерфейсу

Тип *соответствует* (удовлетворяет) интерфейсу, если он обладает всеми методами, которые требует интерфейс. Например, `*os.File` соответствует интерфейсам `io.Reader`, `Writer`, `Closer` и `ReadWriter`. `*bytes.Buffer` соответствует интерфейсам `Reader`, `Writer` и `ReadWriter`, но не `Closer`, потому что не имеет метода `Close`. Для краткости программисты Go часто говорят о том, что конкретный тип "является" типом некоторого интерфейса. Это означает, что он соответствует этому интерфейсу. Например, `*bytes.Buffer` является `io.Writer`, а `*os.File` является `io.ReadWriter`.

Правило присваиваемости (раздел 2.4.2) интерфейсов очень простое: выражение может быть присвоено интерфейсу, только если его тип соответствует этому интерфейсу. Таким образом:

```
var w io.Writer
w = os.Stdout         // OK: *os.File имеет метод Write
w = new(bytes.Buffer) // OK: *bytes.Buffer имеет метод Write
w = time.Second       // Ошибка: у time.Duration нет метода Write

var rwc io.ReadWriteCloser
rwc = os.Stdout         // OK: *os.File имеет методы Read, Write, Close
rwc = new(bytes.Buffer) // Ошибка: у *bytes.Buffer нет метода Close
```

Это правило применимо даже тогда, когда правая сторона сама по себе является интерфейсом:

```
w = rwc // OK: io.ReadWriteCloser имеет метод Write
rwc = w // Ошибка: io.Writer не имеет метода Close
```

Поскольку `ReadWriter` и `ReadWriteCloser` включают все методы `Writer`, любой тип, соответствующий `ReadWriter` или `ReadWriteCloser`, обязательно соответствует `Writer`.

Прежде чем мы пойдем дальше, рассмотрим одно тонкое место, связанное со смыслом утверждения, что некоторый тип имеет некий метод. Вспомним из раздела 6.2, что для каждого именованного конкретного типа `T` одни из его методов имеют получатель типа `T`, в то время как другие требуют указатель `*T`. Вспомним также, что вполне законным является вызов метода `*T` с аргументом типа `T` при условии, что аргумент является *переменной*; компилятор неявно берет ее адрес. Но это лишь синтаксическое упрощение: значение типа `T` не обладает всеми методами, которыми обладает указатель `*T`, и в результате оно может удовлетворить меньшему количеству интерфейсов.

Чтобы было понятнее, приведем пример. Методу `String` типа `IntSet` из раздела 6.5 необходим получатель, являющийся указателем, так что мы не можем вызвать этот метод для неадресуемого значения `IntSet`:

```
type IntSet struct { /* ... */ }
func (*IntSet) String() string
var _ = IntSet{}.String() // Ошибка: String требует получатель *IntSet
```

Но можно вызвать его для переменной `IntSet`:

```
var s IntSet
var _ = s.String() // OK: s является переменной; &s имеет метод String
```

Однако, поскольку только `*IntSet` имеет метод `String`, только `*IntSet` соответствует интерфейсу `fmt.Stringer`:

```
var _ fmt.Stringer = &s // OK
var _ fmt.Stringer = s  // Ошибка: у IntSet нет метода String
```

В разделе 12.8 представлена программа, выводящая методы произвольных значений, а команда `godoc analysis=type` (раздел 10.7.4) отображает методы каждого типа и взаимоотношения между интерфейсами и конкретными типами.

Подобно конверту, который является оболочкой, скрывающей письмо, которое в нем хранится, интерфейс обертывает и скрывает конкретный тип и значение, которое он хранит. При этом могут вызываться только методы интерфейса, даже если конкретный тип имеет и другие методы:

```
os.Stdout.Write([]byte("hello")) // OK: *os.File имеет метод Write
os.Stdout.Close()                // OK: *os.File имеет метод Close

var w io.Writer
w = os.Stdout
w.Write([]byte("hello")) // OK: io.Writer имеет метод Write
w.Close()                // Ошибка: io.Writer не имеет метода Close
```

Интерфейс с большим количеством методов, такой как `io.ReadWriter`, больше говорит о значениях, которые он содержит, и устанавливает более высокие требования к типам, которые его реализуют, чем интерфейс с меньшим количеством методов, такой как `io.Reader`. Что же тип `interface{}`, который вообще не имеет методов, говорит нам о конкретных типах, которые ему соответствуют?

Именно так: ничего. Это может казаться бесполезным, но на самом деле тип `interface{}`, который называется *пустым интерфейсом*, является необходимым. Поскольку тип пустого интерфейса не накладывает никаких требований на типы, которые ему соответствуют, пустому интерфейсу можно присвоить любое значение.

```
var any interface{}
any = true
any = 12.34
any = "hello"
any = map[string]int{"one": 1}
any = new(bytes.Buffer)
```

Хотя это и не очевидно, мы использовали тип пустого интерфейса, начиная с первого примера этой книги, потому что именно он позволяет таким функциям, как `fmt.Println` или `errorf` из раздела 5.7, принимать аргументы любого типа.

Конечно, создав значение `interface{}`, содержащее булев тип, тип с плавающей точкой, строку, отображение, указатель или любой другой тип, мы ничего не можем сделать с ним непосредственно, поскольку интерфейс не имеет методов. Нам нужен способ вновь получить значение переданного типа. Как это сделать с помощью *декларации типа*, вы узнаете в разделе 7.10.

Поскольку удовлетворение интерфейсу зависит только от методов двух типов, нет необходимости объявлять отношения между конкретным типом и интерфейсом, которому он соответствует. Тем не менее иногда полезно документировать и проверять взаимоотношения, если они запланированы, но не могут быть обеспечены программой иным способом. Объявление, приведенное ниже, проверяет во время компиляции, что значение типа `*bytes.Buffer` соответствует интерфейсу `io.Writer`:

```
// *bytes.Buffer должен соответствовать io.Writer
var w io.Writer = new(bytes.Buffer)
```

Нам не нужно выделять новую переменную, поскольку любое значение типа `*bytes.Buffer` будет это делать, даже `nil`, который мы записываем с помощью явного преобразования как `(*bytes.Buffer)(nil)`. А так как мы вообще не намерены обращаться к переменной `w`, ее можно заменить пустым идентификатором. Все эти изменения дают нам этот более скромный вариант:

```
// *bytes.Buffer должен соответствовать io.Writer
var _ io.Writer = (*bytes.Buffer)(nil)
```

Непустым типам интерфейсов, таким как `io.Writer`, чаще всего соответствуют типы указателей, особенно когда один или несколько методов интерфейса подразумевают некоторые изменения получателя, как метод `Write`. Особенно часто используется указатель на структуру.

Однако типы указателей — это отнюдь не единственные типы, которые соответствуют интерфейсам, и даже интерфейсы с изменяющими получателя методами могут быть удовлетворены одним из других ссылочных типов Go. Мы видели примеры типов срезов с методами (`geometry.Path`, раздел 6.1) и типов отображений с методами (`url.Values`, раздел 6.2.1), а позже мы увидим типы функций с методами (`http.HandlerFunc`, раздел 7.7). Даже фундаментальные типы могут соответствовать интерфейсам; как мы видели в разделе 7.4, тип `time.Duration` соответствует интерфейсу `fmt.Stringer`.

Конкретный тип может соответствовать множеству не связанных интерфейсов. Рассмотрим программу, которая организует или продает оцифрованные произведения культуры, такие как музыка, фильмы и книги. Она может определить следующее множество конкретных типов:

```
Album
Book
Movie
Magazine
Podcast
TVEpisode
Track
```

Мы можем выразить каждую интересующую нас абстракцию как интерфейс. Одни свойства являются общими для всех произведений, такие как название, дата создания и список создателей (авторы или художники):

```
type Artifact interface {
    Title()    string
    Creators() []string
    Created()  time.Time
}
```

Другие свойства ограничены только определенными видами произведений. Свойства типографского издания актуальны только для книг и журналов, тогда как только фильмы и телевизионные эпизоды имеют разрешение экрана:

```
type Text interface {
    Pages()    int
    Words()    int
    PageSize() int
}

type Audio interface {
    Stream() (io.ReadCloser, error)
    RunningTime() time.Duration
    Format() string // Например, "MP3", "WAV"
}

type Video interface {
    Stream() (io.ReadCloser, error)
    RunningTime() time.Duration
    Format() string // Например, "MP4", "WMV"
    Resolution() (x, y int)
}
```

Эти интерфейсы представляют собой лишь один из полезных способов группировки связанных конкретных типов и выражения их общих аспектов. Позже мы сможем обнаружить и другие способы группировки. Например, обнаружив, что нужно обрабатывать элементы `Audio` и `Video` одинаково, мы сможем определить интерфейс `Streamer` для представления их общих свойств без изменения каких-либо существующих объявлений типов:

```
type Streamer interface {
    Stream() (io.ReadCloser, error)
    RunningTime() time.Duration
    Format() string
}
```

Каждая группировка конкретных типов на основе их общего поведения может быть выражена как тип интерфейса. В отличие от языков на основе классов, в которых набор интерфейсов, которым соответствует класс, указывается явно, в Go мы можем определять новые абстракции или группы интересов, когда мы в них нуждаемся, не изменяя при этом объявления конкретных типов. Это особенно удобно, когда конкретный тип определен в пакете другого автора. Конечно, при этом должно удовлетворяться требование базовой общности конкретных типов.

7.4. Анализ флагов с помощью `flag.Value`

В этом разделе мы рассмотрим еще один стандартный интерфейс, `flag.Value`, который помогает определить новую запись для флагов командной строки. Рассмотрим приведенную ниже программу, которая бездействует указанный период времени:

```
gopl.io/ch7/sleep
var period = flag.Duration("period", 1*time.Second, "sleep period")

func main() {
    flag.Parse()
    fmt.Printf("Ожидание %v...", *period)
    time.Sleep(*period)
    fmt.Println()
}
```

Перед тем как перейти в режим ожидания, программа выводит соответствующий период времени. Пакет `fmt` вызывает метод `String` типа `time.Duration` для вывода периода не как количества наносекунд, а в удобочитаемом виде:

```
$ go build gopl.io/ch7/sleep
$ ./sleep
Ожидание 1s...
```

По умолчанию период бездействия составляет одну секунду, но им можно управлять с помощью флага командной строки `period`. Функция `flag.Duration` создает переменную типа `time.Duration` и разрешает пользователю указывать продолжительность в различных удобных для человека форматах, включая вывод метода `String`. Такой симметричный дизайн приводит к удобному пользовательскому интерфейсу:

```
$ ./sleep period
50ms
Ожидание 50ms...
$ ./sleep period
2m30s
Ожидание 2m30s...
$ ./sleep period
1.5h
Ожидание 1h30m0s...
$ ./sleep period
"1 day"
неверное значение "1 day" для флага period
```

Поскольку флаги продолжительности очень полезны, эта возможность встроена в пакет `flag`, но можно легко определить новый флаг для собственных типов данных. Нужно только определить тип, который соответствует интерфейсу `flag.Value` и объявление которого показано ниже:

```
package flag

// Value представляет собой интерфейс значения, хранящегося в флаге.
type Value interface {
    String() string
    Set(string) error
}
```

Метод `String` форматирует значение флага для использования в сообщениях справки командной строки; таким образом, каждый `flag.Value` также является `fmt.Stringer`. Метод `Set` анализирует свой строковый аргумент и обновляет значения флага. По сути, метод `Set` является обратным методом для метода `String`, и использовать для них одни и те же обозначения — хорошая практика.

Давайте определим тип `celsiusFlag`, который позволяет указывать температуру в градусах Цельсия или Фаренгейта с соответствующим преобразованием. Обратите внимание, что `celsiusFlag` встраивает `Celsius` (раздел 2.5), тем самым получая метод `String` бесплатно. Чтобы удовлетворить интерфейсу `flag.Value`, осталось только объявить метод `Set`:

gopl.io/ch7/tempconv

```
// *celsiusFlag соответствует интерфейсу flag.Value.
type celsiusFlag struct{ Celsius }

func (f *celsiusFlag) Set(s string) error {
    var unit string
    var value float64
    fmt.Sscanf(s, "%f%s", &value, &unit) // Проверки ошибок не нужны
    switch unit {
    case "C", "°C":
        f.Celsius = Celsius(value)
        return nil
    case "F", "°F":
        f.Celsius = FToC(Fahrenheit(value))
        return nil
    }
    return fmt.Errorf("неверная температура %q", s)
}
```

Вызов `fmt.Sscanf` получает из входной строки `s` число с плавающей точкой (`value`) и строку (`unit`). Хотя обычно необходимо проверить результат вызова `Sscanf` на наличие ошибки, в этом случае такая проверка не нужна, поскольку, если возникнет проблема, в инструкции `switch` просто не найдется нужного соответствия.

Приведенная ниже функция `CelsiusFlag` объединяет все сказанное. Вызывающей функции она возвращает указатель на поле `Celsius`, встроенное в переменную `f` типа `celsiusFlag`. Поле `Celsius` является переменной, которая будет обновляться методом `Set` при обработке флага. Вызов `Var` добавляет флаг в множество флагов командной строки приложения (глобальную переменную `flag.CommandLine`). У программ с необычно сложными интерфейсами командной строки может быть несколько переменных этого типа. Вызов `Var` присваивает аргумент `*celsiusFlag` параметру `flag.Value`, заставляя компилятор выполнить проверку того, что тип `*celsiusFlag` имеет необходимые методы.

```
// CelsiusFlag определяет флаг Celsius с указанным именем, значением
// по умолчанию и строкой-инструкцией по применению и возвращает адрес
// переменной-флага. Аргумент флага должен содержать числовое значение
```

```
// и единицу измерения, например "100C".
func CelsiusFlag(name string, value Celsius, usage string) *Celsius {
    f := celsiusFlag{value}
    flag.CommandLine.Var(&f, name, usage)
    return &f.Celsius
}
```

Теперь можно начать использовать новый флаг в своих программах:

```
gopl.io/ch7/tempflag
var temp = tempconv.CelsiusFlag("temp", 20.0, "температура")

func main() {
    flag.Parse()
    fmt.Println(*temp)
}
```

Вот типичные результаты работы программы (в переводе):

```
$ go build gopl.io/ch7/tempflag
$ ./tempflag
20°C
$ ./tempflag -temp -18C
-18°C
$ ./tempflag -temp 212°F
100°C
$ ./tempflag -temp 273.15K
неверное значение "273.15K" для флага -temp:
    неверная температура "273.15K"
Использование ./tempflag:
    -temp value
        температура (по умолчанию 20°C)
$ ./tempflag help
Использование ./tempflag:
    -temp value
        температура (по умолчанию 20°C)
```

Упражнение 7.6. Добавьте в `tempflag` поддержку температуры по шкале Кельвина.

Упражнение 7.7. Поясните, почему в сообщении о применении программы содержится указание единиц измерения `°C`, в то время как в значении по умолчанию `20.0` единицы измерения не указаны.

7.5. Значения интерфейсов

Концептуально значение интерфейсного типа, или просто *значение интерфейса*, имеет два компонента — конкретный тип и значение этого типа. Они называются *динамическим типом* и *динамическим значением* интерфейса.

Для статически типизированного языка, такого как Go, типы являются понятием времени компиляции, поэтому тип не является значением. В нашей концептуальной модели множество значений, именуемых *дескрипторами типов*, предоставляет сведения о каждом типе, такие как его имя и методы. В значении интерфейса компонент типа представлен соответствующим дескриптором типа.

В инструкциях ниже переменная `w` принимает три разные значения. (Начальное и конечное значения переменной одинаковы.)

```
var w io.Writer
w = os.Stdout
w = new(bytes.Buffer)
w = nil
```

Давайте взглянем на значение и динамическое поведение `w` после каждой инструкции. Первая инструкция объявляет `w`:

```
var w io.Writer
```

Переменные в Go всегда инициализируются точно определенным значением, и интерфейсы не являются исключением. Нулевое значение для интерфейса имеет и тип, и значение, равные `nil` (рис. 7.1).

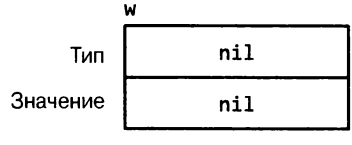


Рис. 7.1. Нулевое значение интерфейса

Значение интерфейса описывается как нулевое или ненулевое на основе его динамического типа, так что здесь вы видите нулевое значение интерфейса. Проверить, является ли значение интерфейса нулевым, можно с помощью инструкций `w == nil` или `w != nil`. Вызов любого метода нулевого интерфейса приводит к аварийной ситуации:

```
w.Write([]byte("hello")) // Авария: разыменование нулевого указателя
```

Вторая инструкция присваивает переменной `w` значение типа `*os.File`:

```
w = os.Stdout
```

Это присваивание включает неявное преобразование из конкретного типа в тип интерфейса, эквивалентное явному преобразованию `io.Writer(os.Stdout)`. Преобразование этого рода, явное или неявное, охватывает тип и значение своего операнда. Динамический тип значения интерфейса устанавливается равным дескриптору для типа указателя `*os.File`, а его динамическое значение хранит копию `os.Stdout`, которая является указателем на переменную `os.File`, представляющую стандартный вывод процесса (рис. 7.2).

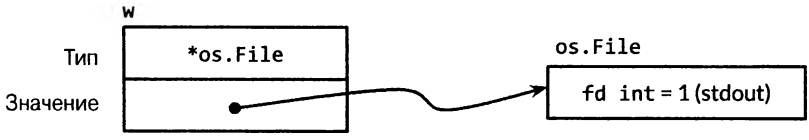


Рис. 7.2. Значение интерфейса, содержащее указатель `*os.File`

Вызов метода `Write` значения интерфейса, содержащего указатель `*os.File`, приводит к вызову метода `(*os.File).Write`. Этот вызов выводит строку `"hello"`.

```
w.Write([]byte("hello")) // "hello"
```

В общем случае во время компиляции мы не можем знать, каким будет динамический тип значения интерфейса, поэтому вызов с помощью интерфейса должен использовать *динамическую диспетчеризацию*. Вместо непосредственного вызова компилятор должен генерировать код для получения адреса метода с именем `Write` из дескриптора типа и выполнить косвенный вызов по этому адресу. Аргументом получателя для вызова является копия динамического значения интерфейса `os.Stdout`. Мы получаем такой же результат, как если бы мы выполнили этот вызов непосредственно:

```
os.Stdout.Write([]byte("hello")) // "hello"
```

Третья инструкция присваивает значение типа `*bytes.Buffer` значению интерфейса:

```
w = new(bytes.Buffer)
```

Теперь динамический тип представляет собой `*bytes.Buffer`, а динамическое значение представляет собой указатель на вновь выделенный буфер (рис. 7.3).

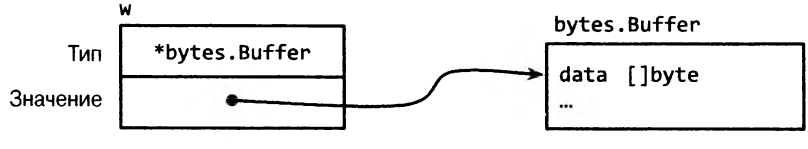


Рис. 7.3. Значение интерфейса, содержащее указатель `*bytes.Buffer`

Вызов метода `Write` использует тот же механизм, который был описан ранее:

```
w.Write([]byte("hello")) // Выводит "hello" в bytes.Buffer
```

На этот раз дескриптором типа является `*bytes.Buffer`, так что вызывается метод `(*bytes.Buffer).Write` с адресом буфера в качестве значения параметра получателя. Вызов добавляет в буфер строку `"hello"`.

Наконец четвертая инструкция присваивает `nil` значению интерфейса:

```
w = nil
```

Эта инструкция сбрасывает оба компонента в нуль, восстанавливая то же состояние `w`, которое было при ее объявлении и которое показано на рис. 7.1.

Значение интерфейса может хранить динамические значения произвольного размера. Например, тип `time.Time`, представляющий момент времени, является структурным типом с несколькими неэкспортируемыми полями. Если мы создадим из него значение интерфейса

```
var x interface{} = time.Now()
```

то результат может выглядеть так, как показано на рис. 7.4. Концептуально динамическое значение всегда размещается внутри значения интерфейса, независимо от того, насколько велик размер его типа. (Это только концептуальная модель; реальная реализация несколько иная.)

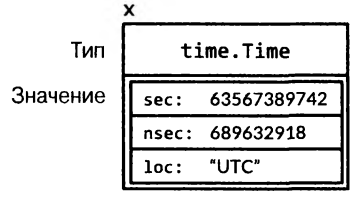


Рис. 7.4. Значение интерфейса, хранящее структуру `time.Time`

Значения интерфейсов можно сравнивать с использованием операторов `==` и `!=`. Два значения интерфейсов равны, если оба равны `nil` или если их динамические типы одинаковы, а динамические значения равны согласно результату сравнения с помощью оператора `==` с обычным поведением для данного типа. Поскольку значения интерфейсов сравнимы, они могут использоваться в качестве ключей отображений или операндов инструкции `switch`.

Однако если сравниваются два значения интерфейсов, имеющих одинаковые динамические типы и эти типы не сравнимы (например, срезы), то сравнение заканчивается аварийной ситуацией:

```
var x interface{} = []int{1, 2, 3}
fmt.Println(x == x) // Аварийная ситуация: несравниваемые типы []int
```

В этом отношении типы интерфейсов необычны. Другие типы безопасно сравниваемы (такие, как фундаментальные типы или указатели) или несравниваемы вообще (срезы, отображения, функции), но при сравнении значений интерфейсов или составных типов, содержащих значения интерфейсов, мы должны учитывать потенциальную возможность возникновения аварийной ситуации. Аналогичный риск существует при использовании интерфейсов в качестве ключей отображений или операндов `switch`. Сравнивайте значения интерфейсов, только если вы уверены, что они содержат динамические значения сравниваемых типов.

При обработке ошибок или при отладке часто оказывается полезной информация о динамическом типе значения интерфейса. Для этого можно использовать символы преобразования `%T` пакета `fmt`:

```
var w io.Writer
fmt.Printf("%T\n", w) // "<nil>"
```

```
w = os.Stdout
fmt.Printf("%T\n", w) // "*os.File"

w = new(bytes.Buffer)
fmt.Printf("%T\n", w) // "*bytes.Buffer"
```

Для получения имени динамического типа интерфейса пакет `fmt` использует рефлексию. Мы познакомимся с ней в главе 12, "Рефлексия".

7.5.1. Осторожно: интерфейс, содержащий нулевой указатель, не является нулевым

Нулевое значение интерфейса, которое не содержит значения как такового, не совпадает со значением интерфейса, содержащим нулевой указатель. Это тонкое различие создает ловушки, в которые попадал каждый программист на Go.

Рассмотрим приведенную ниже программу. Если значение `debug` равно `true`, функция `main` накапливает вывод функции `f` в `bytes.Buffer`.

```
const debug = true

func main() {
    var buf *bytes.Buffer
    if debug {
        buf = new(bytes.Buffer) // Накопление вывода
    }
    f(buf) // Внимание: тонкая ошибка!
    if debug {
        // ... используем buf ...
    }
}

// Если out ненулевой, вывод записывается в него
func f(out io.Writer) {
    // ... некоторые действия ...
    if out != nil {
        out.Write([]byte("выполнено!\n"))
    }
}
```

Можно ожидать, что при изменении значения `debug` на `false` накопление вывода будет отключено, но на самом деле такое изменение приводит к аварийной ситуации во время вызова `out.Write`:

```
if out != nil {
    out.Write([]byte("выполнено!\n")) // Аварийная ситуация:
                        // разыменование нулевого указателя
}
```

Когда `main` вызывает `f`, выполняется присваивание нулевого указателя типа `*bytes.Buffer` параметру `out`, так что динамическое значение `out` равно `nil`.

Однако его динамическим типом является `*bytes.Buffer`, а это означает, что `out` представляет собой ненулевой интерфейс, содержащий нулевое значение указателя (рис. 7.5), так что защитная проверка `out != nil` оказывается пройденной.

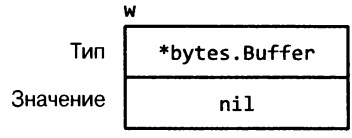


Рис. 7.5. Ненулевой интерфейс, содержащий нулевой указатель

Как и ранее, механизм динамической диспетчеризации определяет, что должен быть вызван `(*bytes.Buffer).Write`, но на этот раз со значением получателя, равным `nil`. Для некоторых типов, таких как `*os.File`, `nil` является допустимым получателем (раздел 6.2.1), но `*bytes.Buffer` среди них нет. Этот метод вызывается, но при попытке получить доступ к буферу возникает аварийная ситуация.

Проблема заключается в том, что хотя нулевой указатель `*bytes.Buffer` имеет методы, необходимые для удовлетворения интерфейсу, он не соответствует *поведенческим* требованиям интерфейса. В частности, вызов нарушает неявное предусловие `(*bytes.Buffer).Write` о том, что получатель не является нулевым, так что присваивание нулевого указателя интерфейсу ошибочное. Решение заключается в том, чтобы изменить тип `buf` в функции `main` на `io.Writer`, избегая, таким образом, присваивания дисфункционального значения интерфейсу:

```
var buf io.Writer
if debug {
    buf = new(bytes.Buffer) // Накопление вывода
}
f(buf) // OK
```

Теперь, когда мы рассмотрели механику значений интерфейсов, давайте взглянем на некоторые более важные интерфейсы из стандартной библиотеки Go. В следующих трех разделах мы увидим, как интерфейсы используются для сортировки, вебслужб и обработки ошибок.

7.6. Сортировка с помощью `sort.Interface`

Как и форматирование строк, сортировка очень часто используется во множестве программ. Хотя минимальная быстрая сортировка может поместиться в 15 строк, надежная реализация существенно больше, и это не тот код, который хотелось бы писать заново или копировать всякий раз, когда он нам нужен.

К счастью, пакет `sort` предоставляет сортировку "на месте" (без привлечения дополнительной памяти) любой последовательности в соответствии с любой функцией упорядочения. Ее дизайн несколько необычен. Во многих языках алгоритм сортировки связан с типом данных последовательности, в то время как функция упорядочения

связана с типом элементов. Напротив, в Go функция `sort.Sort` ничего не предполагает о представлении последовательности или ее элементах. Вместо этого она использует интерфейс, `sort.Interface`, чтобы задать контракт между обобщенным алгоритмом сортировки и всеми типами последовательностей, которые могут быть отсортированы. Реализация этого интерфейса определяет как конкретное представление последовательности, которая часто является срезом, так и желаемый порядок его элементов.

Алгоритм сортировки "на месте" требует трех вещей — длины последовательности, средства сравнения двух элементов и способа обмена двух элементов местами. Все они являются методами `sort.Interface`:

```
package sort

type Interface interface {
    Len() int
    Less(i, j int) bool // i,j – индексы элементов в последовательности
    Swap(i, j int)
}
```

Для сортировки любой последовательности нужно определить тип, который реализует три указанных метода, а затем применить `sort.Sort` к экземпляру этого типа. Как один из простейших примеров рассмотрим сортировку среза строк. Новый тип `StringSlice` и его методы `Len`, `Less` и `Swap` показаны ниже:

```
type StringSlice []string

func (p StringSlice) Len() int { return len(p) }
func (p StringSlice) Less(i, j int) bool { return p[i] < p[j] }
func (p StringSlice) Swap(i, j int) { p[i], p[j] = p[j], p[i] }
```

Теперь мы можем сортировать срез строк, `names`, следующим образом, преобразуя срез в `StringSlice`:

```
sort.Sort(StringSlice(names))
```

Преобразование типа дает значение среза с теми же длиной, емкостью и базовым массивом, что и у `names`, но с типом, который имеет три метода, необходимые для сортировки.

Сортировка среза строк настолько распространена, что пакет `sort` предоставляет тип `StringSlice` и функцию `Strings`, так что показанный выше вызов можно упростить до `sort.Strings(names)`.

Показанная здесь методика легко адаптируется для других сортировок, например, игнорирующих регистр букв или специальные символы. Для более сложной сортировки мы используем ту же идею, но с более сложными структурами данных или более сложными реализациями методов `sort.Interface`.

Рассмотрим еще один пример — программу для сортировки списка воспроизведения музыки, выводимого в виде таблицы. Каждая дорожка представлена отдельной строкой, а каждый столбец представляет один из атрибутов этой дорожки — исполни-

теля, названия и другую информацию. Представьте себе, что таблица выводится с помощью графического пользовательского интерфейса, в котором щелчок на заголовке столбца приводит к сортировке списка по указанному атрибуту; повторный щелчок выполняет сортировку в обратном порядке. Давайте посмотрим, какой может быть реакция на каждый щелчок.

Переменная `tracks` содержит список воспроизведения. (Один из авторов книги приносит извинения за музыкальные вкусы другого автора.) Каждый элемент является косвенным — указателем на `Track`. Хотя приведенный ниже код будет работать и в том случае, если мы будем хранить элементы `Track` непосредственно, но так как функция сортировки меняет местами много пар элементов, она будет работать быстрее, если каждый элемент является указателем (одно машинное слово), а не `Track` полностью (восемь или более слов):

gopl.io/ch7/sorting

```
type Track struct {
    Title  string
    Artist string
    Album  string
    Year   int
    Length time.Duration
}
var tracks = []*Track{
    {"Go", "Delilah", "From the Roots Up", 2012, length("3m38s")},
    {"Go", "Moby", "Moby", 1992, length("3m37s")},
    {"Go Ahead", "Alicia Keys", "As I Am", 2007, length("4m36s")},
    {"Ready 2 Go", "Martin Solveig", "Smash", 2011, length("4m24s")},
}

func length(s string) time.Duration {
    d, err := time.ParseDuration(s)
    if err != nil {
        panic(s)
    }
    return d
}
```

Функция `printTracks` выводит список воспроизведения в виде таблицы. Графический вывод выглядел бы красивее, но в нашей небольшой программе мы воспользовались пакетом `text/tabwriter` для генерации таблицы с красиво выровненными столбцами, как показано ниже. Заметим, что `*tabwriter.Writer` соответствует интерфейсу `io.Writer`. Он накапливает все записанные в него данные; метод `Flush` этого типа форматирует всю таблицу и выводит ее на `os.Stdout`.

```
func printTracks(tracks []*Track) {
    const format = "%v\t%v\t%v\t%v\t%v\t\n"
    tw := new(tabwriter.Writer).Init(os.Stdout, 0, 8, 2, ' ', 0)
    fmt.Fprintf(tw, format, "Title","Artist","Album","Year","Length")
    fmt.Fprintf(tw, format, "-----","------","-----","----","------")
```

```
    for _, t := range tracks {
        fmt.Fprintf(tw, format, t.Title, t.Artist,
                    t.Album, t.Year, t.Length)
    }
    tw.Flush() // Вычисление размеров столбцов и вывод таблицы
}
```

Для сортировки списка воспроизведения по полю `Artist` мы определяем новый тип среза с необходимыми методами `Len`, `Less` и `Swap`, аналогично тому, как мы делали это для `StringSlice`:

```
type byArtist []*Track

func (x byArtist) Len() int { return len(x) }
func (x byArtist) Less(i,j int) bool { return x[i].Artist<x[j].Artist }
func (x byArtist) Swap(i,j int) { x[i], x[j] = x[j], x[i] }
```

Для вызова обобщенной подпрограммы сортировки преобразуем `tracks` в новый тип, `byArtist`, который определяет порядок сортировки:

```
sort.Sort(byArtist(tracks))
```

После сортировки среза по исполнителям вывод `printTracks` имеет следующий вид:

```
Title        Artist          Album              Year  Length
-----        ------          -----              ----  ------
Go Ahead     Alicia Keys     As I Am            2007  4m36s
Go           Delilah         From the Roots Up  2012  3m38s
Ready 2 Go   Martin Solveig  Smash              2011  4m24s
Go           Moby            Moby               1992  3m37s
```

Если пользователь запросит сортировку по исполнителям повторно, будет выполнена сортировка в обратном порядке. Однако нам не нужно определять новый тип `byReverseArtist` с методом `Less`, имеющим обратный смысл, поскольку пакет `sort` предоставляет функцию `Reverse`, которая просто преобразует любой порядок сортировки в обратный:

```
sort.Sort(sort.Reverse(byArtist(tracks)))
```

После сортировки среза по исполнителям в обратном порядке вывод `printTracks` приобретает следующий вид:

```
Title        Artist          Album              Year  Length
-----        ------          -----              ----  ------
Go           Moby            Moby               1992  3m37s
Ready 2 Go   Martin Solveig  Smash              2011  4m24s
Go           Delilah         From the Roots Up  2012  3m38s
Go Ahead     Alicia Keys     As I Am            2007  4m36s
```

Функция `sort.Reverse` заслуживает более близкого знакомства, так как она использует композицию (раздел 6.3), которая является важной идеей. Пакет `sort` опре-

деляет неэкспортируемый тип `reverse`, являющийся структурой, в которую встроен `sort.Interface`. Метод `Less` для `reverse` вызывает метод `Less` встроенного значения `sort.Interface`, но с индексами в обратном порядке, что приводит к обращению результата сортировки:

```
package sort
type reverse struct{ Interface }     // т.е. sort.Interface
func (r reverse) Less(i,j int) bool { return r.Interface.Less(j,i) }
func Reverse(data Interface) Interface { return reverse{data} }
```

Два других метода `reverse`, `Len` и `Swap`, неявно предоставляются исходным значением `sort.Interface`, которое является встроенным полем. Экспортируемая функция `Reverse` возвращает экземпляр типа `reverse`, который содержит исходное значение `sort.Interface`.

Для сортировки по другому столбцу необходимо определить новый тип, такой как `byYear`:

```
type byYear []*Track
func (x byYear) Len() int { return len(x) }
func (x byYear) Less(i, j int) bool { return x[i].Year < x[j].Year }
func (x byYear) Swap(i, j int) { x[i], x[j] = x[j], x[i] }
```

После сортировки `tracks` по году с использованием `sort.Sort(byYear (tracks))`, `printTracks` выводит список в хронологическом порядке:

```
Title        Artist          Album              Year  Length
-----        ------          -----              ----  ------
Go           Moby            Moby               1992  3m37s
Go Ahead     Alicia Keys     As I Am            2007  4m36s
Ready 2 Go   Martin Solveig  Smash              2011  4m24s
Go           Delilah         From the Roots Up  2012  3m38s
```

Для каждого необходимого нам типа элементов среза и каждой функции упорядочения мы объявляем новую реализацию `sort.Interface`. Как вы можете видеть, методы `Len` и `Swap` имеют одинаковые определения для всех типов срезов. В следующем примере конкретный тип `customSort` сочетает срез с функцией, позволяя нам определить новый порядок сортировки с помощью написания одной только функции сравнения. Кстати, конкретные типы, реализующие `sort.Interface`, не всегда являются срезами; `customSort` — это структурный тип:

```
type customSort struct {
    t []*Track
    less func(x, y *Track) bool
}
func (x customSort) Len() int { return len(x.t) }
func (x customSort) Less(i,j int) bool { return x.less(x.t[i],x.t[j]) }
func (x customSort) Swap(i,j int) { x.t[i], x.t[j] = x.t[j], x.t[i] }
```

Давайте определим функцию многоуровневого упорядочения, первичным ключом сортировки которой является `Title`, вторичным ключом является `Year`, а третич-

ным — продолжительность исполнения `Length`. Вот как выглядит вызов `Sort` с использованием анонимной функции упорядочения:

```
sort.Sort(customSort{tracks, func(x, y *Track) bool {
    if x.Title != y.Title {
        return x.Title < y.Title
    }
    if x.Year != y.Year {
        return x.Year < y.Year
    }
    if x.Length != y.Length {
        return x.Length < y.Length
    }
    return false
}})
```

А вот какой вид имеет результат сортировки. Обратите внимание, что две дорожки с одинаковым названием "Go" упорядочены по году выхода:

```
Title        Artist          Album              Year  Length
-----        ------          -----              ----  ------
Go           Moby            Moby               1992  3m37s
Go           Delilah         From the Roots Up  2012  3m38s
Go Ahead     Alicia Keys     As I Am            2007  4m36s
Ready 2 Go   Martin Solveig  Smash              2011  4m24s
```

Хотя сортировка последовательности длиной n требует $O(n \log n)$ операций сравнения, проверка, отсортирована ли данная последовательность, требует не более $n-1$ сравнения. Функция `IsSorted` из пакета `sort` выполняет указанную проверку. Подобно `sort.Sort`, она абстрагирует как последовательность, так и ее функцию упорядочения с помощью `sort.Interface`, но никогда не вызывает метод `Swap`. Приведенный ниже код демонстрирует функции `IntsAreSorted` и `Ints` и тип `IntSlice`:

```
values := []int{3, 1, 4, 1}
fmt.Println(sort.IntsAreSorted(values)) // "false"
sort.Ints(values)
fmt.Println(values)                     // "[1 1 3 4]"
fmt.Println(sort.IntsAreSorted(values)) // "true"
sort.Sort(sort.Reverse(sort.IntSlice(values)))
fmt.Println(values)                     // "[4 3 1 1]"
fmt.Println(sort.IntsAreSorted(values)) // "false"
```

Для удобства пакет `sort` предоставляет версии своих функций и типов, специализированные для `[]int`, `[]string` и `[]float64` с использованием их естественного порядка. Для других типов, таких как `[]int64` или `[]uint`, надо писать собственный, пусть и достаточно короткий, код.

Упражнение 7.8. Многие графические интерфейсы предоставляют таблицы с многоуровневой сортировкой с сохранением состояния: первичный ключ определяется по последнему щелчку на заголовке, вторичный — по предпоследнему и т.д. Опре-

делите реализацию `sort.Interface` для использования в такой таблице. Сравните этот подход с многократной сортировкой с использованием `sort.Stable`.

Упражнение 7.9. Воспользуйтесь пакетом `html/template` (раздел 4.6) для замены `printTracks` функцией, которая выводит дорожки в виде таблицы HTML. Используйте решение предыдущего упражнения для того, чтобы каждый щелчок на заголовке столбца генерировал HTTP-запрос на сортировку таблицы.

Упражнение 7.10. Тип `sort.Interface` можно адаптировать для других применений. Напишите функцию `IsPalindrome(s sort.Interface) bool`, которая сообщает, является ли последовательность `s` палиндромом (другими словами, что обращение последовательности не изменяет ее). Считайте, что элементы с индексами `i` и `j` равны, если `!s.Less(i,j)&&!s.Less(j,i)`.

7.7. Интерфейс `http.Handler`

В главе 1, "Учебник", мы получили представление о том, как использовать пакет `net/http` для реализации веб-клиента (раздел 1.5) и сервера (раздел 1.7). В этом разделе мы детальнее рассмотрим API сервера, в основе которого лежит интерфейс `http.Handler`:

net/http
```
package http

type Handler interface {
    ServeHTTP(w ResponseWriter, r *Request)
}

func ListenAndServe(address string, h Handler) error
```

Функция `ListenAndServe` требует адрес сервера, такой как `"localhost:8000"`, и экземпляр интерфейса `Handler`, которому диспетчеризуются все запросы. Он работает бесконечно, если только не происходит ошибка (или при запуске сервера происходит сбой, так что он не запускается), и в этом случае функция всегда возвращает ненулевую ошибку.

Представьте себе сайт электронного магазина с базой данных, отображающей товары на их цены в долларах. Показанная ниже программа представляет собой простейшую его реализацию. Она моделирует склад как тип отображения, `database`, к которому присоединен метод `ServeHTTP`, так что он соответствует интерфейсу `http.Handler`. Обработчик обходит отображение и выводит его элементы.

gopl.io/ch7/http1
```
func main() {
    db := database{"shoes": 50, "socks": 5}
    log.Fatal(http.ListenAndServe("localhost:8000", db))
}

type dollars float32
```

```
func (d dollars) String() string { return fmt.Sprintf("$%.2f", d) }

type database map[string]dollars

func (db database) ServeHTTP(w http.ResponseWriter,req *http.Request){
    for item, price := range db {
        fmt.Fprintf(w, "%s: %s\n", item, price)
    }
}
```

Если мы запустим сервер

```
$ go build gopl.io/ch7/http1
$ ./http1 &
```

и подключимся к нему с помощью программы `fetch` из раздела Section 1.5 (или, если вам так больше нравится, с помощью веб-браузера), то получим следующий вывод:

```
$ go build gopl.io/ch1/fetch
$ ./fetch http://localhost:8000
shoes: $50.00
socks: $5.00
```

Пока что сервер только перечисляет все товары и отвечает этим на любой запрос, независимо от URL. Более реалистичный сервер определяет несколько различных URL, каждый из которых приводит к своему поведению. Давайте будем вызывать имеющееся поведение при запросе `/list` и добавим еще один запрос `/price`, который сообщает о цене конкретного товара, указанного в параметре запроса наподобие `/price?item=socks`:

gopl.io/ch7/http2

```
func (db database) ServeHTTP(w http.ResponseWriter,req *http.Request){
    switch req.URL.Path {
    case "/list":
        for item, price := range db {
            fmt.Fprintf(w, "%s: %s\n", item, price)
        }
    case "/price":
        item := req.URL.Query().Get("item")
        price, ok := db[item]
        if !ok {
            w.WriteHeader(http.StatusNotFound) // 404
            fmt.Fprintf(w, "нет товара: %q\n", item)
            return
        }
        fmt.Fprintf(w, "%s\n", price)
    default:
        w.WriteHeader(http.StatusNotFound) // 404
        fmt.Fprintf(w, "нет страницы: %s\n", req.URL)
    }
}
```

Теперь обработчик на основе компонента пути URL `req.URL.Path` решает, какая логика должна быть выполнена. Если обработчик не распознает путь, он сообщает клиенту об ошибке HTTP клиента путем вызова `w.WriteHeader(http.StatusNotFound)`; это должно быть сделано до записи любого текста в `w`. (Кстати, `http.ResponseWriter` представляет собой еще один интерфейс. Он дополняет `io.Writer` методами для отправки HTTP-заголовков ответа.) Мы могли бы с тем же результатом использовать и вспомогательную функцию `http.Error`:

```
msg := fmt.Sprintf("нет страницы: %s\n", req.URL)
http.Error(w, msg, http.StatusNotFound)    // 404
```

В случае запроса `/price` вызывается метод `URL.Query`, который выполняет преобразование параметров HTTP-запроса в отображение, а точнее — в мультиотображение типа `url.Values` (раздел 6.2.1) из пакета `net/url`. Затем находится первый параметр `item` и выясняется его цена. Если товар не найден, выводится сообщение об ошибке.

Вот пример сеанса работы с новым сервером:

```
$ go build gopl.io/ch7/http2
$ go build gopl.io/ch1/fetch
$ ./http2 &
$ ./fetch http://localhost:8000/list
shoes: $50.00
socks: $5.00
$ ./fetch http://localhost:8000/price?item=socks
$5.00
$ ./fetch http://localhost:8000/price?item=shoes
$50.00
$ ./fetch http://localhost:8000/price?item=hat
нет товара: "hat"
$ ./fetch http://localhost:8000/help
нет страницы: /help
```

Очевидно, что мы могли бы продолжать добавлять разные варианты действий в `ServeHTTP`, но в реальных приложениях удобнее определить логику для каждого случая в виде отдельной функции или метода. Кроме того, связанным URL может потребоваться схожая логика, например несколько файлов изображений могут иметь URL вида `/images/*.png`. По этим причинам `net/http` предоставляет `ServeMux`, *мультиплексор запросов*, упрощающий связь между URL и обработчиками. `ServeMux` собирает целое множество обработчиков `http.Handler` в единый `http.Handler`. И вновь мы видим, что различные типы, соответствующие одному и тому же интерфейсу, являются *взаимозаменяемыми*: веб-сервер может диспетчеризовать запросы любому `http.Handler`, независимо от того, какой конкретный тип скрывается за ним.

В более сложных приложениях для обработки более сложных требований к диспетчеризации несколько `ServeMux` могут объединяться. Go не имеет канонического веб-каркаса, аналогичного Rails в Ruby или Django в Python. Это не значит, что такого каркаса не может существовать, но строительные блоки в стандартной библиотеке Go

являются столь гибкими, что конкретный каркас просто не нужен. Кроме того, хотя наличие каркаса на ранних этапах проекта удобно, связанные с ним дополнительные сложности могут усложнить долгосрочную поддержку проекта.

В приведенной ниже программе мы создаем `ServeMux` и используем его для сопоставления URL с соответствующими обработчиками для операций `/list` и `/price`, которые были разделены на отдельные методы. Затем мы используем `ServeMux` как основной обработчик в вызове `ListenAndServe`:

gopl.io/ch7/http3

```
func main() {
    db := database{"shoes": 50, "socks": 5}
    mux := http.NewServeMux()
    mux.Handle("/list", http.HandlerFunc(db.list))
    mux.Handle("/price", http.HandlerFunc(db.price))
    log.Fatal(http.ListenAndServe("localhost:8000", mux))
}

type database map[string]dollars

func (db database) list(w http.ResponseWriter, req *http.Request) {
    for item, price := range db {
        fmt.Fprintf(w, "%s: %s\n", item, price)
    }
}

func (db database) price(w http.ResponseWriter, req *http.Request) {
    item := req.URL.Query().Get("item")
    price, ok := db[item]
    if !ok {
        w.WriteHeader(http.StatusNotFound)     // 404
        fmt.Fprintf(w, "no such item: %q\n", item)
        return
    }
    fmt.Fprintf(w, "%s\n", price)
}
```

Давайте сосредоточимся на двух вызовах `mux.Handle`, которые регистрируют обработчики. В первом `db.list` представляет собой значение-метод (раздел 6.4), т.е. значение типа

```
func(w http.ResponseWriter, req *http.Request)
```

которое при вызове вызывает метод `database.list` со значением получателя `db`. Так что `db.list` является функцией, которая реализует поведение обработчика, но так как у этой функции нет методов, она не может соответствовать интерфейсу `http.Handler` и не может быть передана непосредственно `mux.Handle`.

Выражение `http.HandlerFunc(db.list)` представляет собой преобразование типа, а не вызов функции, поскольку `http.HandlerFunc` является типом. Он имеет следующее определение:

net/http
```
package http

type HandlerFunc func(w ResponseWriter, r *Request)

func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
    f(w, r)
}
```

`HandlerFunc` демонстрирует некоторые необычные возможности механизма интерфейсов Go. Это тип функции, имеющей методы и соответствующей интерфейсу `http.Handler`. Поведением его метода `ServeHTTP` является вызов базовой функции. Таким образом, `HandlerFunc` является адаптером, который позволяет значению-функции соответствовать интерфейсу, когда функция и единственный метод интерфейса имеют одинаковую сигнатуру. По сути, этот трюк позволяет одному типу, такому как `database`, соответствовать интерфейсу `http.Handler` несколькими различными способами: посредством его метода `list`, метода `price` и т.д.

Поскольку регистрация обработчика таким образом является весьма распространенной, `ServeMux` имеет удобный метод `HandleFunc`, который делает это для нас, так что мы можем упростить код регистрации обработчика следующим образом:

gopl.io/ch7/http3a
```
mux.HandleFunc("/list", db.list)
mux.HandleFunc("/price", db.price)
```

Из приведенного выше кода легко видеть, как можно создать программу, в которой есть два разных веб-сервера, прослушивающих различные порты, определяющие различные URL и выполняющие диспетчеризацию различным обработчикам. Необходимо просто создать еще один `ServeMux` и выполнить еще один вызов `ListenAndServe`, возможно, параллельно. Но для большинства программ достаточно и одного веб-сервера. Кроме того, обычно обработчики HTTP определяются во многих файлах приложения, и было бы неудобно, если бы все они должны были быть явно зарегистрированы экземпляром `ServeMux` уровня приложения.

Поэтому для удобства пакет `net/http` предоставляет глобальный экземпляр `ServeMux` с именем `DefaultServeMux` и функциями уровня пакета `http.Handle` и `http.HandleFunc`. Для применения `DefaultServeMux` в качестве основного обработчика сервера не нужно передавать его `ListenAndServe`; нужно передать `nil`.

Основная функция сервера при этом упрощается до

gopl.io/ch7/http4
```
func main() {
    db := database{"shoes": 50, "socks": 5}
    http.HandleFunc("/list", db.list)
    http.HandleFunc("/price", db.price)
    log.Fatal(http.ListenAndServe("localhost:8000", nil))
}
```

Наконец еще одно важное напоминание: как мы упоминали в разделе 1.7, веб-сервер вызывает каждый обработчик в новой go-подпрограмме, так что обработчики должны принимать меры предосторожности, такие как *блокировки* при доступе к переменным, к которым могут обращаться другие go-подпрограммы (включая другие запросы того же обработчика). Мы будем говорить о параллелизме в следующих двух главах.

Упражнение 7.11. Добавьте дополнительные обработчики так, чтобы клиент мог создавать, читать, обновлять и удалять записи базы данных. Например, запрос вида `/update?item=socks&price=6` должен обновлять цену товара в базе данных и сообщать об ошибке, если товар отсутствует или цена некорректна (предупреждение: это изменение вносит в программу параллельное обновление переменных).

Упражнение 7.12. Измените обработчик `/list` так, чтобы его вывод представлял собой таблицу HTML, а не текст. Вам может пригодиться пакет `html/template` (раздел 4.6).

7.8. Интерфейс `error`

С самого начала этой книги мы использовали значения таинственного предопределенного типа `error`, не поясняя, что это такое. На самом деле это просто тип интерфейса с единственным методом, который возвращает сообщение об ошибке:

```
type error interface {
    Error() string
}
```

Простейший способ создания `error` — использовать вызов `errors.New`, который возвращает новое значение `error` для данного сообщения об ошибке. Весь пакет `errors` состоит только из четырех строк:

```
package errors

func New(text string) error { return &errorString{text} }

type errorString struct { text string }


func (e *errorString) Error() string { return e.text }
```

Базовый тип `errorString` является структурой, а не строкой, чтобы защитить его представление от случайного (или умышленного) обновления. Причина, по которой тип указателя `*errorString`, а не сам тип `errorString`, соответствует интерфейсу `error`, заключается в том, что каждый вызов `New` должен создавать экземпляр `error`, не равный никаким другим. Было бы не слишком удобно, если бы, например, ошибка `io.EOF` оказывалась при сравнении равной ошибке, случайно имеющей то же сообщение:

```
fmt.Println(errors.New("EOF") == errors.New("EOF")) // "false"
```

Вызовы `errors.New` сравнительно редки, поскольку имеется удобная функция-оболочка, `fmt.Errorf`, которая заодно выполняет форматирование строки. Мы использовали ее несколько раз в главе 5, "Функции".

```
package fmt

import "errors"

func Errorf(format string, args ...interface{}) error {
    return errors.New(Sprintf(format, args...))
}
```

Хотя `*errorString` может быть простейшим типом `error`, он далеко не единственный. Например, пакет `syscall` предоставляет API низкоуровневых системных вызовов Go. На многих платформах он определяет числовой тип `Errno`, соответствующий интерфейсу `error`, а на платформе Unix метод `Error` типа `Errno` выполняет поиск в таблице строк, как показано ниже:

```
package syscall

type Errno uintptr     // Код ошибки операционной системы
var errors = [...]string{
    1: "операция не разрешена",         // EPERM
    2: "нет такого файла или каталога", // ENOENT
    3: "нет такого процесса",           // ESRCH
    // ...
}

func (e Errno) Error() string {
    if 0 <= int(e) && int(e) < len(errors) {
        return errors[e]
    }
    return fmt.Sprintf("errno %d", e)
}
```

Приведенная далее инструкция создает значение-интерфейс, хранящее значение `Errno`, равное 2 и означающее условие ENOENT POSIX:

```
var err error = syscall.Errno(2)
fmt.Println(err.Error()) // "нет такого файла или каталога"
fmt.Println(err)         // "нет такого файла или каталога"
```

Графически значение `err` показано на рис. 7.6.

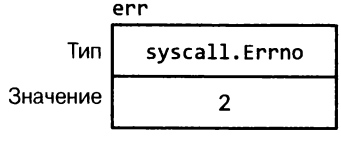


Рис. 7.6. Значение интерфейса, хранящее целочисленное значение `syscall.Errno`

Errno является эффективным представлением ошибок системных вызовов, выбираемых из конечного множества, соответствующим стандартному интерфейсу error. Другие типы, соответствующие этому интерфейсу, мы увидим в разделе 7.11.

7.9. Пример: вычислитель выражения

В этом разделе будет создан вычислитель простых арифметических выражений. Мы используем интерфейс Expr для представления любого выражения на этом языке. Пока что в нем нет методов, но мы добавим их позже:

```
// Expr представляет арифметическое выражение
type Expr interface{}
```

Наш язык выражений состоит из литералов с плавающей точкой; бинарных операторов +, -, * и /; унарных операторов -x и +x; вызовов функций pow(x,y), sin(x) и sqrt(x); переменных, таких как x и pi; и, конечно же, скобок и правил стандартных приоритетов операторов. Все значения имеют тип float64. Вот несколько примеров выражений:

```
sqrt(A / pi)
pow(x, 3) + pow(y, 3)
(F - 32) * 5 / 9
```

Пять приведенных ниже конкретных типов представляют отдельные разновидности выражений. Var представляет ссылку на переменную (вы вскоре поймете, почему она экспортируемая). literal представляет константу с плавающей точкой. Типы unary и binary представляют выражения с операторами с одним или двумя операндами, которые, в свою очередь, могут быть любыми разновидностями Expr. call представляет вызов функции; мы ограничим ее поле fn значениями pow, sin и sqrt.

gopl.io/ch7/eval

```
// Var определяет переменную, например x.
type Var string

// literal представляет собой числовую константу, например 3.141.
type literal float64

// unary представляет выражение с унарным оператором, например -x.
type unary struct {
    op rune // '+' или '-'
    x Expr
}

// binary представляет выражение с бинарным оператором, например x+y.
type binary struct {
    op rune // '+', '-', '*' или '/'
    x, y Expr
}
```

```
// call представляет выражение вызова функции, например sin(x).
type call struct {
    fn string // одно из "pow", "sin", "sqrt"
    args []Expr
}
```

Для вычисления выражений, содержащих переменные, нам понадобится *среда*, которая отображает имена переменных на значения:

```
type Env map[Var]float64
```

Нам также нужно, чтобы каждый вид выражения определял метод `Eval`, который возвращает значение выражения в данной среде. Поскольку каждое выражение должно предоставлять этот метод, мы добавим его к интерфейсу `Expr`. Пакет экспортирует только типы `Expr`, `Env` и `Var`; клиенты могут использовать вычислитель без обращения к другим типам выражений.

```
type Expr interface {
    // Eval возвращает значение данного Expr в среде env.
    Eval(env Env) float64
}
```

Ниже приводятся конкретные методы `Eval`. Метод для `Var` выполняет поиск в среде, который возвращает нуль, если переменная не определена, а метод для `literal` просто возвращает значение литерала.

```
func (v Var) Eval(env Env) float64 {
    return env[v]
}

func (l literal) Eval(_ Env) float64 {
    return float64(l)
}
```

Методы `Eval` для `unary` и `binary` рекурсивно вычисляют их операнды, а затем применяют к ним операцию `op`. Мы не рассматриваем деление на нуль или бесконечность как ошибки, так как они дают результат, хотя и не являющийся конечным. Наконец для `call` метод вычисляет аргументы функций `pow`, `sin` или `sqrt`, а затем вызывает соответствующую функцию из пакета `math`.

```
func (u unary) Eval(env Env) float64 {
    switch u.op {
    case '+':
        return +u.x.Eval(env)
    case '-':
        return -u.x.Eval(env)
    }
    panic(fmt.Sprintf("неподдерживаемый унарный оператор: %q", u.op))
}

func (b binary) Eval(env Env) float64 {
```

```
    switch b.op {
    case '+':
        return b.x.Eval(env) + b.y.Eval(env)
    case '-':
        return b.x.Eval(env) - b.y.Eval(env)
    case '*':
        return b.x.Eval(env) * b.y.Eval(env)
    case '/':
        return b.x.Eval(env) / b.y.Eval(env)
    }
    panic(fmt.Sprintf("неподдерживаемый бинарный оператор: %q", b.op))
}

func (c call) Eval(env Env) float64 {
    switch c.fn {
    case "pow":
        return math.Pow(c.args[0].Eval(env), c.args[1].Eval(env))
    case "sin":
        return math.Sin(c.args[0].Eval(env))
    case "sqrt":
        return math.Sqrt(c.args[0].Eval(env))
    }
    panic(fmt.Sprintf("неподдерживаемый вызов функции: %s", c.fn))
}
```

Некоторые из этих методов могут завершаться ошибкой. Например, выражение `call` может содержать неизвестную функцию или неверное количество аргументов. Можно также создать `unary` или `binary` с недопустимым оператором, таким как ! или < (хотя упомянутая ниже функция `Parse` никогда не поступит таким образом). Эти ошибки приводят к аварийной ситуации в `Eval`. Другие ошибки, такие как вычисление `Var`, отсутствующей в среде, просто заставляют `Eval` вернуть неверный результат. Все эти ошибки могут быть обнаружены путем проверки `Expr` перед его вычислением. В этом и состоит работа метода `Check`, который мы вскоре вам покажем, но сначала давайте проверим `Eval`.

Приведенная ниже функция `TestEval` представляет собой проверку вычислителя. Она использует пакет `testing`, который мы будем рассматривать в главе 11, "Тестирование", но пока достаточно знать, что вызов `t.Errorf` сообщает об ошибке. Функция проходит по таблице входных данных, которая определяет три выражения и различные среды для каждого из них. Первое выражение вычисляет радиус круга по его площади `A`, второе вычисляет сумму кубов двух переменных, `x` и `y`, а третье преобразует температуру по Фаренгейту `F` в температуру по Цельсию.

```
func TestEval(t *testing.T) {
    tests := []struct {
        expr string
        env Env
        want string
    }{
```

```
        {"sqrt(A / pi)", Env{"A": 87616, "pi": math.Pi}, "167"},
        {"pow(x, 3) + pow(y, 3)", Env{"x": 12, "y": 1}, "1729"},
        {"pow(x, 3) + pow(y, 3)", Env{"x": 9, "y": 10}, "1729"},
        {"5 / 9 * (F - 32)", Env{"F": -40}, "-40"},
        {"5 / 9 * (F - 32)", Env{"F": 32}, "0"},
        {"5 / 9 * (F - 32)", Env{"F": 212}, "100"},
    }
    var prevExpr string
    for _, test := range tests {
        // Выводит expr, только когда оно изменяется.
        if test.expr != prevExpr {
            fmt.Printf("\n%s\n", test.expr)
            prevExpr = test.expr
        }
        expr, err := Parse(test.expr)
        if err != nil {
            t.Error(err) // Ошибка анализа
            continue
        }
        got := fmt.Sprintf("%.6g", expr.Eval(test.env))
        fmt.Printf("\t%v => %s\n", test.env, got)
        if got != test.want {
            t.Errorf("%s.Eval() в %v = %q, требуется %q\n",
                     test.expr, test.env, got, test.want)
        }
    }
}
```

Для каждой записи в таблице тест выполняет синтаксический анализ выражения, вычисляет его в определенной среде и выводит результат. У нас нет возможности привести здесь функцию `Parse`, но вы сможете найти ее исходный текст, если загрузите пакет с помощью `go get`.

Команда `go test` (раздел 11.1) запускает тесты пакета:

```
$ go test -v gopl.io/ch7/eval
```

Флаг `-v` позволяет увидеть вывод теста, который обычно подавляется для успешно пройденных тестов. Вот вывод инструкций `fmt.Printf` теста:

```
sqrt(A / pi)
    map[A:87616 pi:3.141592653589793] => 167

pow(x, 3) + pow(y, 3)
    map[x:12 y:1] => 1729
    map[x:9 y:10] => 1729

5 / 9 * (F - 32)
    map[F:-40]=> -40
    map[F:32] => 0
    map[F:212] => 100
```

К счастью, пока что все входные данные корректно сформированы, но вряд ли так будет всегда. Даже в интерпретируемых языках распространена проверка синтаксиса на *статические* ошибки, т.е. на ошибки, которые можно обнаружить без запуска программы. Отделяя статические проверки от динамических, можно быстрее обнаруживать ошибки и выполнять много проверок только один раз, вместо того чтобы выполнять проверки при каждом вычислении выражения.

Давайте добавим в интерфейс `Expr` еще один метод. Метод `Check` выполняет проверку на статические ошибки в синтаксическом дереве выражения. Его параметр `vars` мы поясним чуть позже.

```
type Expr interface {
    Eval(env Env) float64
    // Check сообщает об ошибках в данном Expr и добавляет свои Vars.
    Check(vars map[Var]bool) error
}
```

Ниже показаны конкретные методы `Check`. Вычисление `literal` и `Var` не может быть неудачным, поэтому методы `Check` для этих типов возвращают `nil`. Методы для `unary` и `binary` сначала проверяют, что оператор является допустимым, а затем рекурсивно проверяют операнды. Аналогично метод для `call` сначала проверяет, что функция известна и имеет правильное количество аргументов, а затем рекурсивно проверяет каждый аргумент.

```
func (v Var) Check(vars map[Var]bool) error {
    vars[v] = true
    return nil
}

func (literal) Check(vars map[Var]bool) error {
    return nil
}

func (u unary) Check(vars map[Var]bool) error {
    if !strings.ContainsRune("+-", u.op) {
        return fmt.Errorf("некорректный унарный оператор %q", u.op)
    }
    return u.x.Check(vars)
}

func (b binary) Check(vars map[Var]bool) error {
    if !strings.ContainsRune("+-*/", b.op) {
        return fmt.Errorf("некорректный бинарный оператор %q", b.op)
    }
    if err := b.x.Check(vars); err != nil {
        return err
    }
    return b.y.Check(vars)
}
```

```
func (c call) Check(vars map[Var]bool) error {
    arity, ok := numParams[c.fn]
    if !ok {
        return fmt.Errorf("неизвестная функция %q", c.fn)
    }
    if len(c.args) != arity {
        return fmt.Errorf("вызов %s имеет %d вместо %d аргументов",
            c.fn, len(c.args), arity)
    }
    for _, arg := range c.args {
        if err := arg.Check(vars); err != nil {
            return err
        }
    }
    return nil
}

var numParams = map[string]int{"pow": 2, "sin": 1, "sqrt": 1}
```

Мы разделяем некорректные входные данные и ошибки, которые они вызывают, на две группы. (Не показанная) функция `Parse` сообщает о синтаксических ошибках, а функция `Check` — о семантических.

```
x % 2 непредвиденный символ '%'
math.Pi непредвиденный символ '.'
!true непредвиденный символ '!'
"hello" непредвиденный символ '"'

log(10) неизвестная функция "log"
sqrt(1, 2) вызов sqrt имеет 2 вместо 1 аргументов
```

Аргумент функции `Check`, множество `Var`, накапливает множество имен переменных в выражении. Каждая из этих переменных должна присутствовать в среде для оценки успеха. Этот набор логически является *результатом* вызова функции `Check`, но поскольку метод является рекурсивным, более удобно заполнять множество, передаваемое в качестве параметра. В первоначальном вызове клиент должен предоставить пустое множество.

В разделе 3.2 мы выводили график функции `f(x,y)`, зафиксированной во время компиляции. Теперь, когда можно выполнять синтаксический анализ, проверку и вычисление выражений, заданных в виде строк, мы можем построить веб-приложение, которое получает выражение функции во время выполнения от клиента и строит ее график. Мы можем использовать множество `vars` для проверки, что выражение представляет собой функцию только от двух переменных, `x` и `y` (на самом деле от трех, так как мы для удобства предоставляем возможность использовать радиус `r`). Мы будем использовать метод `Check`, того чтобы отклонить некорректные выражения до их вычисления, (а не повторять эти проверки 40 000 раз (100×100 ячеек, каждая с четырьмя углами) при вычислении значений функции).

Функция `parseAndCheck` объединяет описанные шаги анализа и проверки:

```
gopl.io/ch7/surface
import "gopl.io/ch7/eval"

func parseAndCheck(s string) (eval.Expr, error) {
    if s == "" {
        return nil, fmt.Errorf("пустое выражение")
    }
    expr, err := eval.Parse(s)
    if err != nil {
        return nil, err
    }
    vars := make(map[eval.Var]bool)
    if err := expr.Check(vars); err != nil {
        return nil, err
    }
    for v := range vars {
        if v != "x" && v != "y" && v != "r" {
            return nil, fmt.Errorf("неизвестная переменная: %s", v)
        }
    }
    return expr, nil
}
```

Чтобы получить из этого веб-приложение, все, что нам надо, — это приведенная ниже функция `plot`, которая имеет знакомую сигнатуру `http.HandlerFunc`:

```
func plot(w http.ResponseWriter, r *http.Request) {
    r.ParseForm()
    expr, err := parseAndCheck(r.Form.Get("expr"))
    if err != nil {
        http.Error(w, "некорректное выражение: "+err.Error(),
                   http.StatusBadRequest)
        return
    }
    w.Header().Set("Content-Type", "image/svg+xml")
    surface(w, func(x, y float64) float64 {
        r := math.Hypot(x, y) // Расстояние от (0,0)
        return expr.Eval(eval.Env{"x": x, "y": y, "r": r})
    })
}
```

Функция `plot` выполняет синтаксический анализ и проверку выражения, указанного в запросе HTTP, и использует его для создания анонимной функции от двух переменных. Анонимная функция имеет ту же сигнатуру, что и фиксированная функция `f` в исходной программе черчения графиков, но вычисляет выражение, заданное пользователем. Среда определяет переменные `x` и `y` и радиус `r`. Наконец `plot` вызывает функцию `surface`, которая является просто функцией `main` из `gopl.io/ch3/surface`, измененной таким образом, чтобы получать в качестве параметров функцию черчения и вывод `io.Writer` вместо фиксированной функции `f` и `os.Stdout`. На рис. 7.7 показаны три поверхности, начерченные программой.

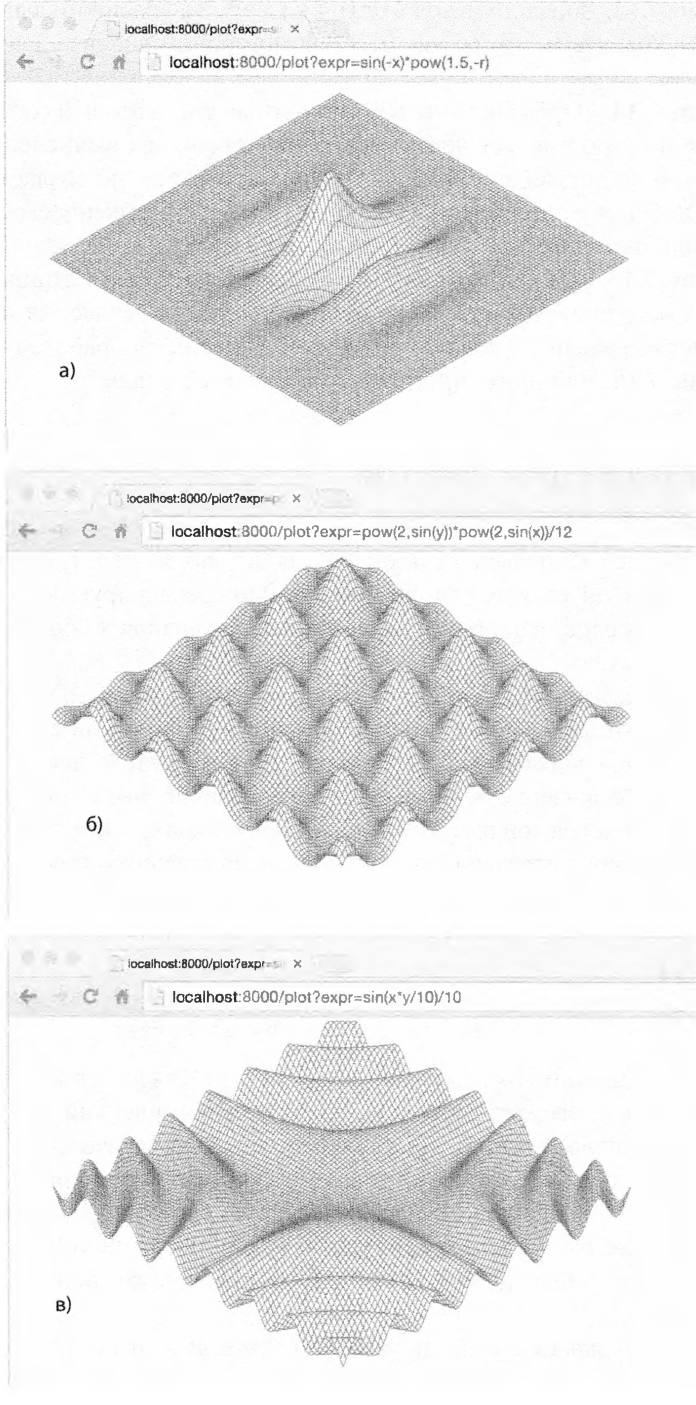


Рис. 7.7. Графики трех функций: а) `sin(-x)*pow(1.5,-r)`; б) `pow(2,sin(y))*pow(2,sin(x))/12`; в) `sin(x*y/10)/10`

Упражнение 7.13. Добавьте метод `String` к `Expr` для красивого вывода синтаксического дерева. Убедитесь, что результаты при повторном анализе дают эквивалентное дерево.

Упражнение 7.14. Определите новый конкретный тип, который соответствует интерфейсу `Expr` и предоставляет новую операцию, такую как вычисление минимального из значений ее операндов. Поскольку функция `Parse` не создает экземпляров этого нового типа, для ее применения вам надо строить синтаксическое дерево непосредственно (или расширять синтаксический анализатор).

Упражнение 7.15. Напишите программу, которая читает из стандартного ввода единственное выражение, предлагает пользователю ввести значения переменных, а затем вычисляет выражение в полученной среде. Аккуратно обработайте все ошибки.

Упражнение 7.16. Напишите программу-калькулятор для веб.

7.10. Декларации типов

Декларация типа (type assertion) представляет собой операцию, применяемую к значению-интерфейсу. Синтаксически она выглядит, как `x.(T)`, где `x` — выражение интерфейсного типа, а `T` является типом, именуемым "декларируемым" (asserted). Декларация типа проверяет, что динамический тип его операнда `x` соответствует декларируемому типу.

Существуют две возможности. Во-первых, если декларируемый тип `T` является конкретным типом, то декларация типов проверяет, *идентичен* ли динамический тип `x` типу `T`. Если эта проверка завершается успешно, результатом декларации типа является динамическое значение `x`, типом которого, конечно, является `T`. Другими словами, декларация типа для конкретного типа извлекает конкретное значение из своего операнда. Если проверка неудачна, то создается аварийная ситуация, например:

```
var w io.Writer
w = os.Stdout
f := w.(*os.File)      // Успешно: f == os.Stdout
c := w.(*bytes.Buffer) // Аварийная ситуация: интерфейс хранит
                       // *os.File, а не *bytes.Buffer
```

Во-вторых, если вместо этого декларированный тип `T` является типом интерфейса, то декларация типов проверяет, *соответствует* ли динамический тип `x` интерфейсу `T`. Если проверка прошла успешно, динамическое значение не извлекается; значением результата остается значение интерфейса с тем же типом и значениями компонентов, но сам результат имеет тип интерфейса `T`. Другими словами, декларация типа для типа интерфейса изменяет тип выражения, делая доступным иной (обычно больший) набор методов, но сохраняет динамический тип и значения компонентов внутри значения интерфейса.

После первой декларации типа, показанной ниже, и `w`, и `rw` хранят `os.Stdout`, так что каждая из этих переменных имеет динамический тип `*os.File`, но `w`, которая является `io.Writer`, демонстрирует только метод `Write`, в то время как `rw` имеет еще и метод `Read`.

```
var w io.Writer
w = os.Stdout
rw := w.(io.ReadWriter) // Успех: *os.File имеет методы Read и Write
w = new(ByteCounter)
rw = w.(io.ReadWriter)  // Аварийная ситуация: *ByteCounter
                        // не имеет метода Read
```

Независимо от того, какой тип был декларирован, если операнд представляет собой интерфейсное значение `nil`, декларация типа является неудачной. Декларация типа для менее ограничивающего типа интерфейса (с меньшим количеством методов) требуется редко, так как ведет себя так же, как присваивание, за исключением случая `nil`:

```
w = rw             // io.ReadWriter присваиваем io.Writer
w = rw.(io.Writer) // Ошибка только при rw == nil
```

Часто мы не уверены в динамическом типе значения интерфейса и хотим проверить, не является ли он некоторым определенным типом. Если декларация типа появляется в присваивании, в котором ожидаются два результата, как, например, в приведенных ниже объявлениях, аварийная ситуация при неудаче операции не происходит. Вместо этого возвращается дополнительный второй результат булева типа, указывающий успех или неудачу операции:

```
var w io.Writer = os.Stdout
f, ok := w.(*os.File)      // Успех:    ok, f == os.Stdout
b, ok := w.(*bytes.Buffer) // Неудача: !ok, b == nil
```

Второй результат присваивается переменной с именем `ok`. Если операция не удалась, `ok` получает значение `false`, а первый результат равен нулевому значению декларированного типа, который в данном примере представляет собой нулевое значение `*bytes.Buffer`.

Результат `ok` часто немедленно используется для принятия решения о последующих действиях. Расширенная форма инструкции `if` позволяет сделать это достаточно компактно:

```
if f, ok := w.(*os.File); ok {
    // ... Использование f ...
}
```

Когда операнд декларации типа представляет собой переменную, вместо другого имени для новой локальной переменной иногда повторно используется исходное имя, затеняющее оригинал:

```
if w, ok := w.(*os.File); ok {
    // ... Использование w ...
}
```

7.11. Распознавание ошибок с помощью деклараций типов

Рассмотрим множество ошибок, возвращаемых файловыми операциями в пакете os. Операция ввода-вывода может завершиться ошибкой из-за множества причин, но зачастую три разновидности неудач должны быть обработаны по-разному: файл уже существует (для операций создания файла), файл не найден (для операций чтения) и отсутствие прав доступа. Пакет os предоставляет три вспомогательные функции для классификации сбоев в соответствии со значением error:

```
package os

func IsExist(err error) bool
func IsNotExist(err error) bool
func IsPermission(err error) bool
```

Наивная реализация этих предикатов может проверять, что сообщение об ошибке содержит определенную подстроку:

```
func IsNotExist(err error) bool {
    // Примечание: не надежно!
    return strings.Contains(err.Error(), "файл не существует")
}
```

Однако, поскольку логика обработки ошибок ввода-вывода может варьироваться от одной платформы к другой, этот подход не является надежным, а одна и та же ошибка может быть передана с различными сообщениями. Проверка наличия подстроки в сообщении об ошибке может оказаться полезной при тестировании, чтобы обеспечить корректное сообщение об ошибке функцией, но для производственного кода его недостаточно.

Более надежный подход заключается в представлении структурированных значений ошибок с помощью специального типа. Пакет os определяет тип PathError для описания сбоев, связанных с операциями над путями файлов, такими как Open или Delete, а также вариант LinkError для описания сбоев операций, включающих два пути к файлам, например операций Symlink или Rename. Вот как выглядит os.PathError:

```
package os

// PathError записывает ошибку, а также операцию и путь к файлу,
// которые ее вызвали.
type PathError struct {
    Op   string
    Path string
    Err  error
}
```

```
func (e *PathError) Error() string {
    return e.Op + " " + e.Path + ": " + e.Err.Error()
}
```

Большинство клиентов не обращают внимания на `PathError` и обрабатывают все ошибки единообразно, путем вызова их методов `Error`. Хотя метод `Error` ошибки `PathError` формирует сообщение с использованием простой конкатенации полей, структура `PathError` сохраняет базовые компоненты ошибки. Клиенты, которым необходимо отличать один вид отказа от другого, могут использовать декларации типа для обнаружения определенного типа ошибки; конкретный тип обеспечивает более подробную информацию, чем простая строка.

```
_, err := os.Open("/no/such/file")
fmt.Println(err)   // "открытие /no/such/file: нет такого файла"
fmt.Printf("%#v\n", err)
// Вывод:
// &os.PathError{Op:"open", Path:"/no/such/file", Err:0x2}
```

Так работают три упомянутые вспомогательные функции. Например, показанная ниже `IsNotExist` сообщает, равна ли ошибка `syscall.ENOENT` (раздел 7.8) или отличной от нее ошибке `os.ErrNotExist` (см. `io.EOF` в разделе 5.4.2), или `*PathError`, базовой ошибкой которой является одна из перечисленных.

```
import (
    "errors"
    "syscall"
)

var ErrNotExist = errors.New("файл не существует")

// IsNotExist возвращает булево значение, указывающее, произошла
// ли ошибка, связанная с отсутствием файла или каталога. Условию
// соответствует как ошибка ErrNotExist, так и некоторые ошибки
// системных вызовов.
func IsNotExist(err error) bool {
    if pe, ok := err.(*PathError); ok {
        err = pe.Err
    }
    return err == syscall.ENOENT || err == ErrNotExist
}
```

А вот как выглядит применение этой функции:

```
_, err := os.Open("/no/such/file")
fmt.Println(os.IsNotExist(err))   // "true"
```

Конечно, структура `PathError` теряется, если сообщение об ошибке объединяется в более крупную строку, например, с помощью вызова `fmt.Errorf`. Обычно распознавание ошибки должно выполняться сразу после сбоя, прежде чем ошибка будет передана вызывающей функции.

7.12. Запрос поведения с помощью деклараций типов

Приведенная ниже логика схожа с частью веб-сервера `net/http`, ответственной за написание полей HTTP-заголовка, таких как `"Content-Type: text/html"`. `io.Writer w` представляет HTTP-ответ; байты, записываемые в него, в конечном итоге отправляются к кому-то на веб-браузер.

```
func writeHeader(w io.Writer, contentType string) error {
    if _, err := w.Write([]byte("Content-Type: ")); err != nil {
        return err
    }
    if _, err := w.Write([]byte(contentType)); err != nil {
        return err
    }
    // ...
}
```

Поскольку метод `Write` требует байтового среза, а значение, которое мы хотим записывать, является строкой, необходимо преобразование `[]byte(...)`. Это преобразование выделяет память и делает копию, но копия почти сразу же после записи отбрасывается. Давайте представим, что эта часть является чуть ли не ядром веб-сервера и что профилирование кода показывает, что такое распределение памяти существенно замедляет работу. Можем ли мы избежать такого распределения памяти?

Интерфейс `io.Writer` говорит нам о конкретном типе, хранящемся в `w`, только то, что в него можно записать байты. Если мы взглянем за кулисы пакета `net/http`, то увидим, что динамический тип, который хранит `w`, имеет также метод `WriteString`, который позволяет нам эффективно записывать строки без создания временной копии. (Это может показаться странным, но ряд важных типов, соответствующих интерфейсу `io.Writer`, имеют также метод `WriteString`, включая такие типы, как `*bytes.Buffer`, `*os.File` и `*bufio.Writer`.)

Мы не можем утверждать, что произвольный `io.Writer w` также имеет метод `WriteString`. Но мы можем определить новый интерфейс, который имеет только этот метод, и использовать декларацию типа для проверки, соответствует ли динамический тип `w` этому новому интерфейсу.

```
// writeString записывает s в w.
// Если w имеет метод WriteString, он вызывается вместо w.Write.
func writeString(w io.Writer, s string) (n int, err error) {
    type stringWriter interface {
        WriteString(string) (n int, err error)
    }
    if sw, ok := w.(stringWriter); ok {
        return sw.WriteString(s) // Избегаем копирования
    }
    return w.Write([]byte(s))    // Используем временную копию
}
```

```
func writeHeader(w io.Writer, contentType string) error {
    if _, err := writeString(w, "Content-Type: "); err != nil {
        return err
    }
    if _, err := writeString(w, contentType); err != nil {
        return err
    }
    // ...
}
```

Чтобы избежать повторения, мы переместили проверку во вспомогательную функцию `writeString`, но она настолько полезна, что стандартная библиотека предоставляет ее как `io.WriteString`. Это рекомендованный способ записи строки в `io.Writer`.

В этом примере любопытно то, что нет стандартного интерфейса, который определяет метод `WriteString` и указывает его требуемое поведение. Кроме того, соответствует ли конкретный тип интерфейсу `stringWriter`, определяется только его методами, а не каким-то объявленным взаимоотношением между ним и типом интерфейса. Это означает, что показанная выше методика опирается на предположение, что *если* тип соответствует показанному ниже интерфейсу, *то* вызов `WriteString(s)` должен выполнять те же действия, что и `Write([]byte(s))`.

```
interface {
    io.Writer
    WriteString(s string) (n int, err error)
}
```

Хотя `io.WriteString` документирует это предположение, несколько функций, которые его вызывают, вероятно, должны документировать, что они делают такое же предположение. Определение метода у некоторого типа является неявным согласием на определенный поведенческий контракт. Новичкам в Go, особенно тем, у которых имеется опыт работы со строго типизированными языками, это отсутствие явного указания намерений может показаться тревожащим, но на практике оно редко вызывает проблемы. За исключением пустого интерфейса `interface{}`, типы интерфейса редко удовлетворяются из-за непреднамеренного совпадения.

Показанная выше функция `writeString` использует декларацию типа для того, чтобы выяснить, соответствует ли значение общего типа интерфейса более конкретному типу интерфейса, и если это так, то она использует поведение последнего. Эта методика может использоваться независимо от того, является ли запрашиваемый интерфейс стандартным, как `io.ReadWriter`, или пользовательским, как `stringWriter`.

Именно так `fmt.Fprintf` отличает значения, соответствующие `error` или `fmt.Stringer`, от всех прочих значений. В `fmt.Fprintf` имеется шаг, преобразующий отдельный операнд в строку, что-то вроде этого:

```
package fmt

func formatOneValue(x interface{}) string {
    if err, ok := x.(error); ok {
```

```
        return err.Error()
    }
    if str, ok := x.(Stringer); ok {
        return str.String()
    }
    // ... Все прочие типы ...
}
```

Если `x` соответствует любому из этих двух интерфейсов, они определяют форматирование данного значения. Если же нет, выполняется более или менее унифицированная обработка всех прочих типов с использованием рефлексии; об этом мы расскажем в главе 12, "Рефлексия".

И вновь здесь используется предположение, что любой тип с методом `String` соответствует контракту поведения `fmt.Stringer`, который возвращает годящуюся для вывода строку.

7.13. Выбор типа

Интерфейсы используются двумя различными способами. В первом из них, примерами которого являются `io.Reader`, `io.Writer`, `fmt.Stringer`, `sort.Interface`, `http.Handler` и `error`, методы интерфейсов выражают подобие конкретных типов, которые соответствуют данному интерфейсу, но скрывают детали представления и внутренние операции этих конкретных типов. Акцент при этом делается на методах, а не на конкретных типах.

Второй стиль использует способность значения интерфейса хранить значения ряда конкретных типов и рассматривает интерфейс как *объединение* этих типов. Декларации типов используются для динамического распознавания этих типов и трактовки каждого отдельного случая по-своему. В этом стиле акцент делается на конкретных типах, соответствующих интерфейсу, а не на методах интерфейса (если он имеет таковые), и нет никакого сокрытия информации. Мы будем говорить об интерфейсах, используемых таким образом, как о *распознаваемых объединениях*.

Если вы знакомы с объектно-ориентированным программированием, то вы можете узнать в этих двух стилях *полиморфизм подтипов* и *перегрузку*, но вам не нужно запоминать эти термины. В оставшейся части главы мы представим примеры второго стиля.

API Go для запросов к базе данных SQL, как и другие языки, позволяет отделить фиксированную часть запроса от переменных частей. Пример клиента может выглядеть следующим образом:

```
import "database/sql"

func listTracks(db sql.DB, artist string, minYear, maxYear int) {
    result, err := db.Exec(
      "SELECT * FROM tracks WHERE artist = ? AND ?<=year AND year<=?",
      artist, minYear, maxYear)
    // ...
}
```

Метод `Exec` заменяет каждый символ `?` в строке SQL-запроса литералом, обозначающим значение соответствующего аргумента, который может быть логическим значением, числом, строкой или иметь значение `nil`. Построение запросов таким образом помогает избежать атак SQL-инъекций, при которых злоумышленник получает контроль над запросом, используя некорректное заключение в кавычки входных данных. В `Exec` мы могли бы найти функцию наподобие показанной ниже, которая преобразует значение каждого аргумента в его SQL-запись в виде литерала.

```
func sqlQuote(x interface{}) string {
    if x == nil {
        return "NULL"
    } else if _, ok := x.(int); ok {
        return fmt.Sprintf("%d", x)
    } else if _, ok := x.(uint); ok {
        return fmt.Sprintf("%d", x)
    } else if b, ok := x.(bool); ok {
        if b {
            return "TRUE"
        }
        return "FALSE"
    } else if s, ok := x.(string); ok {
        return sqlQuoteString(s) // (Функция не показана)
    } else {
        panic(fmt.Sprintf("непредвиденный тип %T: %v", x, x))
    }
}
```

Инструкция `switch` упрощает цепочку `if-else`, которая выполняет последовательность проверок на равенство значений. Аналогичная инструкция *выбора типа* (type switch) упрощает цепочку `if-else` деклараций типов.

В простейшем виде выбор типа выглядит, как обычная инструкция `switch`, в которой операндом является `x.(type)` — здесь `type` представляет собой ключевое слово, — а каждая инструкция `case` имеет один или несколько типов. Такая инструкция обеспечивает множественное ветвление на основе динамического типа значения интерфейса. Случай `nil` соответствует ситуации `x == nil`, а случай `default` обрабатывает ситуацию, когда соответствие не найдено. Выбор типа для `sqlQuote` имеет следующий вид:

```
switch x.(type) {
    case nil:       // ...
    case int, uint: // ...
    case bool:      // ...
    case string:    // ...
    default:        // ...
}
```

Как и в обычной инструкции `switch` (раздел 1.8), все инструкции `case` рассматриваются по порядку, и когда соответствие найдено, выполняется тело соответству-

ющей инструкции case. Порядок инструкций становится важным, когда один или несколько участвующих в сравнениях типов являются интерфейсами, так как при этом возможна ситуация, когда соответствие будет найдено в нескольких инструкциях case. Положение инструкции default относительно прочих инструкций значения не имеет. В выборе типа применение fallthrough не разрешено.

Обратите внимание, что в исходной функции логика для случаев bool и string требует доступа к значению, извлеченному декларацией типа. Так как это достаточно типичная ситуация, инструкция выбора типа имеет расширенную форму, которая в каждом case связывает извлекаемое значение с новой переменной:

```
switch x := x.(type) { /* ... */ }
```

Здесь новая переменная также названа x; как и в случае декларации типа, повторное использование имен переменных является достаточно распространенным. Подобно обычной инструкции switch, инструкция выбора типа неявно создает лексический блок, так что объявление новой переменной x не конфликтует с переменной x во внешнем блоке. Каждый case также неявно создает отдельный лексический блок.

Перепишем sqlQuote с использованием расширенного выбора типов, что делает код существенно понятнее:

```
func sqlQuote(x interface{}) string {
    switch x := x.(type) {
    case nil:
        return "NULL"
    case int, uint:
        return fmt.Sprintf("%d", x) // Здесь x имеет тип interface{}.
    case bool:
        if x {
            return "TRUE"
        }
        return "FALSE"
    case string:
        return sqlQuoteString(x)    // (Не показана)
    default:
        panic(fmt.Sprintf("непредвиденный тип %T: %v", x, x))
    }
}
```

В этой версии, в блоке каждого case с единственным типом переменная x имеет тот же тип, что и указанный в case. Например, x имеет тип bool в case bool и тип string в case string. Во всех остальных случаях x имеет (интерфейсный) тип операнда switch, в данном примере — interface{}. Когда одно и то же действие требуется для нескольких case, таких как int и uint, выбор типа позволяет легко их объединить.

Хотя sqlQuote принимает аргумент любого типа, функция выполняется до конца, только если тип аргумента соответствует одному из case в инструкции выбора типа; в противном случае осуществляется аварийная ситуация с сообщением “непредви-

денный тип". Хотя типом x является interface{}, мы рассматриваем его как *распознаваемое объединение* int, uint, bool, string и nil.

7.14. Пример: XML-декодирование на основе лексем

В разделе 4.5 было показано, как декодировать документы JSON в структуры Go с помощью функций Marshal и Unmarshal из пакета encoding/json. Пакет encoding/xml предоставляет аналогичный API. Этот подход удобен, когда мы хотим построить представление дерева документа, но для многих программ это не требуется. Пакет encoding/xml также предоставляет для декодирования XML низкоуровневый API *на основе лексем*. При использовании стиля на основе лексем синтаксический анализатор получает входные данные и генерирует поток лексем, главным образом четырех видов (StartElement, EndElement, CharData и Comment), каждый из которых является конкретным типом пакета encoding/xml. Каждый вызов (*xml.Decoder).Token возвращает лексему.

Ниже приведены существенные для нашего рассмотрения части API:

```
encoding/xml
package xml

type Name struct {
    Local string // Например, "Title" или "id"
}

type Attr struct { // Например, name="value"
    Name Name
    Value string
}

// Token включает StartElement, EndElement, CharData
// и Comment, а также некоторые скрытые типы (не показаны).
type Token interface{}
type StartElement struct { // Например, <name>
    Name Name
    Attr []Attr
}
type EndElement struct { Name Name } // Например, </name>
type CharData []byte                 // Например, <p>CharData</p>
type Comment []byte                  // Например, <!-- Comment -->

type Decoder struct{ /* ... */ }
func NewDecoder(io.Reader) *Decoder
func (*Decoder)Token()(Token, error) // Возвращает очередную лексему
```

Не имеющий методов интерфейс `Token` также является примером распознаваемого объединения. Предназначение традиционного интерфейса наподобие `io.Reader` — скрыть детали конкретных типов, которые ему соответствуют, так, чтобы могли быть созданы новые реализации; все конкретные типы обрабатываются одинаково. Напротив, набор конкретных типов, которые соответствуют распознаваемому объединению, является изначально фиксированным и не скрытым. Типы распознаваемого объединения имеют несколько методов; функции, которые работают с ними, выражаются в виде набора `case` выбора типа, с различной логикой в каждом конкретном случае.

Приведенная далее программа `xmlselect` извлекает и выводит текст определенных элементов дерева XML-документа. С помощью описанного выше API она может делать свою работу за один проход по входным данным, без построения дерева.

gopl.io/ch7/xmlselect

```
// Xmlselect выводит текст выбранных элементов XML-документа.
package main

import (
    "encoding/xml"
    "fmt"
    "io"
    "os"
    "strings"
)

func main() {
    dec := xml.NewDecoder(os.Stdin)
    var stack []string // Стек имен элементов
    for {
        tok, err := dec.Token()
        if err == io.EOF {
            break
        } else if err != nil {
            fmt.Fprintf(os.Stderr, "xmlselect: %v\n", err)
            os.Exit(1)
        }
        switch tok := tok.(type) {
        case xml.StartElement:
            stack = append(stack, tok.Name.Local) // Внесение в стек
        case xml.EndElement:
            stack = stack[:len(stack)-1]          // Снятие со стека
        case xml.CharData:
            if containsAll(stack, os.Args[1:]) {
                fmt.Printf("%s: %s\n", strings.Join(stack, " "), tok)
            }
        }
    }
}
```

```
// containsAll указывает, содержит ли x элементы y в том же порядке.
func containsAll(x, y []string) bool {
    for len(y) <= len(x) {
        if len(y) == 0 {
            return true
        }
        if x[0] == y[0] {
            y = y[1:]
        }
        x = x[1:]
    }
    return false
}
```

Каждый раз, когда цикл в `main` встречает `StartElement`, он помещает имя элемента в стек, а для каждого `EndElement` снимает имя со стека. API гарантирует, что лексемы `StartElement` и `EndElement` в последовательности будут корректно соответствовать друг другу даже в неверно сформированных документах. Комментарии игнорируются. Когда программа `xmlselect` обнаруживает `CharData`, она выводит текст, только если стек содержит все элементы с именами, перечисленными в качестве аргументов командной строки в том же порядке.

Показанная ниже команда выводит текст всех элементов `h2`, которые находятся ниже двух уровней элементов `div`. Вводом программы является спецификация XML, сама являющаяся XML-документом.

```
$ go build gopl.io/ch1/fetch
$ ./fetch http://www.w3.org/TR/2006/RECxml1120060816 |
    ./xmlselect div div h2
html body div div h2: 1 Introduction
html body div div h2: 2 Documents
html body div div h2: 3 Logical Structures
html body div div h2: 4 Physical Structures
html body div div h2: 5 Conformance
html body div div h2: 6 Notation
html body div div h2: A References
html body div div h2: B Definitions for Character Normalization
...
```

Упражнение 7.17. Расширьте возможности `xmlselect` так, чтобы элементы могли быть выбраны не только по имени, но и по атрибутам, в духе CSS, так что, например, элемент наподобие `<div id="page" class="wide">` может быть выбран как по соответствию атрибутов `id` или `class`, так и по его имени.

Упражнение 7.18. С помощью API декодера на основе лексем напишите программу, которая будет читать произвольный XML-документ и строить представляющее его дерево. Узлы могут быть двух видов: узлы `CharData` представляют текстовые строки, а узлы `Element` — именованные элементы и их атрибуты. Каждый узел элемента имеет срез дочерних узлов.

Вам могут пригодиться следующие объявления:

```
import "encoding/xml"

type Node interface{} // CharData or *Element

type CharData string

type Element struct {
    Type       xml.Name
    Attr     []xml.Attr
    Children []Node
}
```

7.15. Несколько советов

Проектирование нового пакета начинающие программисты Go часто начинают с создания множества интерфейсов и только после этого определяют конкретные типы, которые им соответствуют. Этот подход дает несколько интерфейсов, каждый из которых имеет только одну реализацию. Не делайте этого. Такие интерфейсы являются ненужными абстракциями, которые к тому же имеют свою ненулевую стоимость во время выполнения. Ограничить методы типа или поля структуры, видимые извне пакета, можно с помощью механизма экспорта (раздел 6.6). Интерфейсы нужны только в том случае, если есть несколько конкретных типов, работа с которыми должна выполняться единообразно.

Исключением из этого правила является ситуация, когда интерфейсу соответствует единственный конкретный тип, но этот тип не может находиться в том же пакете, что и интерфейс, из-за своих зависимостей. В таком случае интерфейс является хорошим средством развязывания двух пакетов.

Поскольку интерфейсы используются в Go только тогда, когда им соответствуют два или более типов, они обязательно абстрагируются от деталей любой конкретной реализации. В результате получаются интерфейсы меньшего размера с небольшими, простыми методами, причем зачастую только с одним методом, как в случае с `io.Writer` или `fmt.Stringer`. Новым типам проще соответствовать небольшим интерфейсам. При проектировании интерфейса стоит следовать правилу *запрашивайте только самое необходимое*.

На этом наше знакомство с методами и интерфейсами завершено. Go поддерживает объектно-ориентированный стиль программирования, но это не означает, что вам нужно использовать исключительно его. Не все должно быть объектом; свою нишу занимают автономные функции, так же как и не инкапсулированные типы данных. Обратите внимание, что все примеры в первых пяти главах этой книги вместе вызывают не более двух десятков методов, таких как `input.Scan`, в отличие от обычных функций наподобие `fmt.Printf`.

8

Go-подпрограммы и каналы

Параллельное программирование, выражение программы как композиции ряда автономных действий, никогда не было столь важным, как сегодня. Веб-серверы одновременно обрабатывают запросы тысяч клиентов. Приложения на планшетах и в телефонах визуализируют анимацию интерфейса пользователя и одновременно в фоновом режиме выполняют вычисления и сетевые запросы. Даже традиционные пакетные задачи — чтение некоторых входных данных, выполнение вычислений и запись некоторых выходных данных — используют параллелизм для того, чтобы скрыть задержки операций ввода-вывода и использовать множество процессоров современного компьютера, количество которых с каждым годом растет все больше, в отличие от их скорости.

Go обеспечивает два стиля параллельного программирования. В этой главе представлены go-подпрограммы (goroutines) и каналы, которые поддерживают *взаимодействующие последовательные процессы* (communicating sequential processes — CSP), модель параллелизма, в которой между независимыми процессами (go-подпрограммами) передаются значения, но переменные по большей части ограничиваются одним процессом. В главе 9, "Параллельность и совместно используемые переменные", охвачены некоторые аспекты более традиционной модели *многопоточности с общей памятью*, которая будет вам знакома, если вы использовали потоки в других основных языках программирования. Там же описаны важные опасности и ловушки параллельного программирования, в которые мы не будем углубляться в данной главе.

Несмотря на то что поддержка параллелизма в Go является одной из его самых сильных сторон, рассмотрение параллельных программ существенно труднее, чем последовательных, а приобретенная при последовательном программировании интуиция может время от времени вводить в заблуждение. Если это ваша первая встреча с параллелизмом, мы рекомендуем потратить немного дополнительного времени на серьезные размышления над приводимыми в этих двух главах примерами.

8.1. Go-подпрограммы

В Go каждая одновременно выполняющаяся задача называется *go-подпрограммой* (goroutine). Рассмотрим программу, в которой есть две функции, одна из которых выполняет некоторые вычисления, а вторая записывает некоторые выходные данные,

и предположим, что ни одна из них не вызывает другую. Последовательная программа может вызвать одну функцию, а затем вторую, но в *параллельной* программе с двумя или более go-подпрограммами вызовы *обеих* функций могут быть активны в одно и то же время. Мы вскоре увидим такую программу.

Если вы использовали потоки операционной системы или потоки в других языках, то пока можете считать, что go-подпрограмма схожа с потоком, и сразу же писать корректные параллельные программы. Различия между потоками и go-подпрограммами по существу количественные, а не качественные, и описаны в разделе 9.8.

При запуске программы ее единственной go-подпрограммой является та, которая вызывает функцию `main`, поэтому мы называем ее *главной go-подпрограммой*. Новые go-подпрограммы создаются с помощью инструкции `go`. Синтаксически инструкция `go` является вызовом обычной функции или метода, которому предшествует ключевое слово `go`. Инструкция `go` заставляет функцию быть вызванной во вновь созданной go-подпрограмме. Сама по себе инструкция `go` немедленно завершается:

```
f()    // Вызов f(); ожидание возврата из нее
go f() // Создание новой go-подпрограммы, вызывающей f(); ожидания нет
```

В приведенном ниже примере главная go-подпрограмма вычисляет 45-е число Фибоначчи. Поскольку она использует ужасно неэффективный рекурсивный алгоритм, время ее работы весьма значительно, и мы хотели бы предоставить пользователю визуальное подтверждение того, что программа все еще работает, отображая анимированный текстовый элемент, представляющий собой вращающийся отрезок.

gopl.io/ch8/spinner

```
func main() {
    go spinner(100 * time.Millisecond)
    const n = 45
    fibN := fib(n) // Медленное вычисление
    fmt.Printf("\rFibonacci(%d) = %d\n", n, fibN)
}

func spinner(delay time.Duration) {
    for {
        for _, r := range `-\|/`
        {
            fmt.Printf("\r%c", r)
            time.Sleep(delay)
        }
    }
}

func fib(x int) int {
    if x < 2 {
        return x
    }
    return fib(x-1) + fib(x-2)
}
```

После нескольких секунд анимации вызов `fib(45)` завершается, и функция `main` выводит свой результат:

```
Fibonacci(45) = 1134903170
```

После этого функция `main` завершается. Когда это происходит, все go-подпрограммы тут же прекращают выполнение, и программа завершает свою работу. Помимо возврата из функции `main` или выхода из программы, нет никакого программного способа остановить из одной go-подпрограммы другую, но, как мы увидим позже, есть способы обмена информацией с go-подпрограммой, с помощью которых можно попросить go-подпрограмму остановиться самостоятельно.

Обратите внимание, как программа выражена в виде композиции двух автономных процессов, анимации "пиктограммы" и вычисления числа Фибоначчи. Каждый из них записан как отдельная функция, но обе они работают одновременно.

8.2. Пример: параллельный сервер часов

Сети — естественная область применения параллелизма, поскольку серверы обычно обрабатывают много клиентских подключений одновременно, при этом каждый клиент, по существу, независим от других. В этом разделе мы познакомимся с пакетом `net`, который предоставляет компоненты для построения программ сетевых клиентов и серверов, сообщающихся посредством TCP, UDP или сокетов Unix. Пакет `net/http`, который мы уже использовали в книге, является надстройкой над функциями пакета `net`.

Наш первый пример — последовательный сервер часов, который выводит текущее время клиенту один раз в секунду:

gopl.io/ch8/clock1

```
// Clock1 является TCP-сервером, периодически выводящим время.
package main
import (
    "io"
    "log"
    "net"
    "time"
)

func main() {
    listener, err := net.Listen("tcp", "localhost:8000")
    if err != nil {
        log.Fatal(err)
    }
    for {
        conn, err := listener.Accept()
        if err != nil {
            log.Print(err) // Например, обрыв соединения
            continue
```

```
        }
        handleConn(conn)   // Обработка единственного подключения
    }
}

func handleConn(c net.Conn) {
    defer c.Close()
    for {
        _, err := io.WriteString(c, time.Now().Format("15:04:05\n"))
        if err != nil {
            return // Например, отключение клиента
        }
        time.Sleep(1 * time.Second)
    }
}
```

Функция `Listen` создает объект `net.Listener`, который прослушивает входящие соединения на сетевом порту, в данном случае это TCP-порт `localhost:8000`. Метод `Accept` прослушивателя блокируется до тех пор, пока не будет сделан входящий запрос на подключение, после чего возвращает объект `net.Conn`, представляющий соединение.

Функция `handleConn` обрабатывает одно полное клиентское соединение. Она в цикле выводит клиенту текущее временя, `time.Now()`. Поскольку `net.Conn` соответствует интерфейсу `io.Writer`, мы можем осуществлять вывод непосредственно в него. Цикл завершается, когда выполнение записи не удается, например потому, что клиент был отключен, и при этом `handleConn` закрывает свою сторону соединения с помощью отложенного вызова `Close` и переходит в состояние ожидания очередного запроса на подключение.

Метод `time.Time.Format` предоставляет способ форматирования значений даты и времени на основе примера. Его аргументом является шаблон, указывающий способ форматирования времени, в частности — `Mon Jan 2 03:04:05PM 2006 UTC0700`. Момент времени имеет восемь компонентов (день недели, месяц, день месяца и т.д.). В строке `Format` может появиться любое их подмножество, в любом порядке и в различных форматах; выбранные компоненты даты и времени будут отображаться в выбранном формате. Здесь мы просто используем часы, минуты и секунды времени. Пакет `time` определяет шаблоны для многих стандартных форматов времени, таких как `time.RFC1123`. Такой же механизм используется и в обратном направлении в ходе анализа строки времени с помощью `time.Parse`.

Для подключения к серверу нам понадобится клиентская программа, такая как `nc` ("netcat"), стандартная вспомогательная программа для работы с сетевыми подключениями:

```
$ go build gopl.io/ch8/clock1
$ ./clock1 &
$ nc localhost 8000
13:58:54
```

```
13:58:55
13:58:56
13:58:57
^C
```

Клиент отображает время, отправляемое сервером каждую секунду до тех пор, пока мы не прервем работу клиента с помощью комбинации клавиш <Ctrl+C>, которая в Unix-системах выводится оболочкой как ^C. Если `nc` или `netcat` в вашей системе не установлен, можно использовать `telnet` или простую Go-версию `netcat`, которая использует `net.Dial` для подключения к TCP-серверу:

gopl.io/ch8/netcat1

```
// Netcat1 – TCP-клиент только для чтения.
package main

import (
    "io"
    "log"
    "net"
    "os"
)

func main() {
    conn, err := net.Dial("tcp", "localhost:8000")
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()
    mustCopy(os.Stdout, conn)
}

func mustCopy(dst io.Writer, src io.Reader) {
    if _, err := io.Copy(dst, src); err != nil {
        log.Fatal(err)
    }
}
```

Эта программа считывает данные из подключения и записывает их в стандартный поток вывода до тех пор, пока не будет достигнут конец файла или не произойдет ошибка. Функция `mustCopy` является вспомогательной функцией, используемой в нескольких примерах данного раздела. Давайте теперь запустим два клиента одновременно на разных терминалах, которые показаны столбцами слева и справа:

```
$ go build gopl.io/ch8/netcat1
$ ./netcat1
13:58:54                                    $ ./netcat1
13:58:55
13:58:56
^C
```

```
                                                    13:58:57
                                                    13:58:58
                                                    13:58:59
                                                    ^C
$ killall clock1
```

Команда `killall` в Unix прекращает работу всех процессов с данным именем.

Второй клиент вынужден ждать завершения работы первого клиента, поскольку сервер является *последовательным*; он занимается только одним клиентом одновременно. Внесем одно небольшое изменение для того, чтобы сделать сервер параллельным: добавление ключевого слова `go` к вызову `handleConn` приводит к тому, что каждый вызов осуществляется в собственной go-подпрограмме.

gopl.io/ch8/clock2

```
for {
    conn, err := listener.Accept()
    if err != nil {
        log.Print(err)  // Например, разрыв соединения
        continue
    }
    go handleConn(conn) // Параллельная обработка соединений
}
```

Теперь получать значение времени могут несколько клиентов одновременно:

```
$ go build gopl.io/ch8/clock2
$ ./clock2 &
$ go build gopl.io/ch8/netcat1
$ ./netcat1
14:02:54                                            $ ./netcat1
14:02:55                                            14:02:55
14:02:56                                            14:02:56
14:02:57                                            ^C
14:02:58
14:02:59                                            $ ./netcat1
14:03:00                                            14:03:00
14:03:01                                            14:03:01
^C                                                  14:03:02
                                                    ^C
$ killall clock2
```

Упражнение 8.1. Измените программу `clock2` таким образом, чтобы она принимала номер порта, и напишите программу `clockwall`, которая действует в качестве клиента нескольких серверов одновременно, считывая время из каждого и выводя результаты в виде таблицы, сродни настенным часам, которые можно увидеть в некоторых офисах. Если у вас есть доступ к географически разнесенным компьютерам, запустите экземпляры серверов удаленно; в противном случае запустите локальные экземпляры на разных портах с поддельными часовыми поясами.

```
$ TZ=US/Eastern      ./clock2 -port 8010 &
$ TZ=Asia/Tokyo      ./clock2 -port 8020 &
$ TZ=Europe/London   ./clock2 -port 8030 &
$ clockwall NewYork=localhost:8010
↳     London=localhost:8030 Tokyo=localhost:8020
```

Упражнение 8.2. Реализуйте параллельный FTP-сервер. Сервер должен интерпретировать команды от каждого клиента, такие как `cd` для изменения каталога, `ls` для вывода списка файлов в каталоге, `get` для отправки содержимого файла и `close` для закрытия соединения. В качестве клиента можно использовать стандартную команду `ftp` или написать собственную программу.

8.3. Пример: параллельный эхо-сервер

Сервер часов использует по одной go-подпрограмме на соединение. В этом разделе мы создадим эхо-сервер, который для каждого подключения использует несколько go-подпрограмм. Большинство эхо-серверов просто записывают все, что считывают, — что можно выполнить с помощью следующей тривиальной версии `handleConn`:

```
func handleConn(c net.Conn) {
    io.Copy(c, c)   // Примечание: игнорируем ошибки
    c.Close()
}
```

Более интересный эхо-сервер может имитировать реверберацию обычного эха, сначала отвечая громко (`"HELLO!"`), затем, после задержки, — умеренно (`"Hello!"`), а потом — совсем тихо (`"hello!"`). Это умеет делать следующая версия `handleConn`:

gopl.io/ch8/reverb1

```
func echo(c net.Conn, shout string, delay time.Duration) {
    fmt.Fprintln(c, "\t", strings.ToUpper(shout))
    time.Sleep(delay)
    fmt.Fprintln(c, "\t", shout)
    time.Sleep(delay)
    fmt.Fprintln(c, "\t", strings.ToLower(shout))
}

func handleConn(c net.Conn) {
    input := bufio.NewScanner(c)
    for input.Scan() {
        echo(c, input.Text(), 1*time.Second)
    }
    // Примечание: игнорируем потенциальные ошибки input.Err()
    c.Close()
}
```

Нашу клиентскую программу необходимо обновить так, чтобы она отправляла входные данные на сервер и в то же время копировала ответ сервера на выход, что предоставляет еще одну возможность использования параллелизма:

gopl.io/ch8/netcat2

```
func main() {
    conn, err := net.Dial("tcp", "localhost:8000")
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()
    go mustCopy(os.Stdout, conn)
    mustCopy(conn, os.Stdin)
}
```

В то время как главная go-подпрограмма считывает стандартный ввод и отправляет его на сервер, вторая go-подпрограмма считывает и выводит ответ сервера. Когда главная go-подпрограмма встречает конец входных данных, например после того, как пользователь нажмет в терминале клавиши <Ctrl+D> (или <Ctrl+Z> в Microsoft Windows), программа останавливается, даже если другие go-подпрограммы все еще работают. (Познакомившись с каналами разделе 8.4.1, мы узнаем, как заставить программу ждать завершения с обеих сторон соединения.)

В приведенной ниже сессии входные данные клиента приведены слева, а ответы сервера — справа. Клиент три раза "кричит" эхо-серверу и слушает ответы.

```
$ go build gopl.io/ch8/reverb1
$ ./reverb1 &
$ go build gopl.io/ch8/netcat2
$ ./netcat2
Hello?
            HELLO?
            Hello?
            hello?
Is there anybody there?
            IS THERE ANYBODY THERE?
Yooohooo!
            Is there anybody there?
            is there anybody there?
            YOOOHOOO!
            Yooohooo!
            yooohooo!
^D
$ killall reverb1
```

Обратите внимание, что третий "крик" от клиента не рассматривается до тех пор, пока второй крик не обработан полностью, — что, конечно, не очень реалистично. Реальное эхо будет состоять из трех независимых криков. Чтобы имитировать его,

нам понадобится больше go-подпрограмм. И вновь все, что нам нужно сделать, — это добавить ключевое слово go, на этот раз к вызову echo:

gopl.io/ch8/reverb2

```
func handleConn(c net.Conn) {
    input := bufio.NewScanner(c)
    for input.Scan() {
        go echo(c, input.Text(), 1*time.Second)
    }
    // Примечание: игнорируем потенциальные ошибки input.Err()
    c.Close()
}
```

Аргументы функции, запускаемой с помощью go, вычисляются при выполнении самой инструкции go; таким образом, input.Text() вычисляется в главной go-подпрограмме.

Теперь все эхо работают параллельно и перекрываются во времени:

```
$ go build gopl.io/ch8/reverb2
$ ./reverb2 &
$ ./netcat2
Is there anybody there?
            IS THERE ANYBODY THERE?
Yooohooo!
            Is there anybody there?
            YOOOHOOO!
            is there anybody there?
            Yooohooo!
            yooohooo!
^D
$ killall reverb2
```

Все, что требовалось для того, чтобы сделать сервер использующим параллелизм, причем не только для обработки соединений от нескольких клиентов, но даже в рамках одного соединения, — это вставить два ключевых слова go.

Однако, добавляя эти ключевые слова, мы должны были убедиться, что этот одновременный вызов методов net.Conn безопасен, что для большинства типов не выполняется. Мы обсудим решающее значение концепции *безопасности параллелизма* в следующей главе.

8.4. Каналы

Если go-подпрограммы представляют собой процессы в рамках параллельной программы Go, то каналы являются соединениями между ними. Канал представляет собой механизм связи, который позволяет одной go-подпрограмме отправлять некоторые значения другой go-подпрограмме. Каждый канал является средством переда-

чи значений определенного типа, который называется *типом элементов* канала. Тип канала, элементы которого имеют тип `int`, записывается как `chan int`.

Для создания канала мы используем встроенную функцию `make`:

```
ch := make(chan int) // ch имеет тип 'chan int'
```

Как и в случае с отображениями, канал является *ссылкой* на структуру данных, создаваемую с помощью функции `make`. Копируя канал или передавая его функции в качестве аргумента, мы копируем ссылку, поэтому вызывающая и вызываемая функции ссылаются на одну и ту же структуру данных. Как и в случае с другими ссылочными типами, нулевым значением канала является `nil`.

Два канала одного типа можно сравнивать с помощью оператора `==`. Сравнение имеет значение `true`, если оба канала являются ссылками на одну и ту же структуру данных канала. Канал также можно сравнить со значением `nil`.

Канал имеет две основные операции, *отправление* и *получение*, вместе известные как *коммуникации* (communications). Инструкция отправления передает через канал значение из одной go-подпрограммы другой, выполняющей соответствующее выражение получения. Обе операции записываются с использованием оператора `<-`. В инструкции отправления `<-` разделяет канал и значение. В выражении получения `<-` предшествует операнду канала. Выражение получения, результат которого не используется, является корректной инструкцией.

```
ch <- x  // Инструкция отправления
x = <-ch // Выражение получения в инструкции присваивания
<-ch     // Инструкция получения; результат не используется
```

Каналы поддерживают третью операцию, *закрытие*, которая устанавливает флаг, указывающий, что по этому каналу больше не будут передаваться значения; после этого любые попытки отправления завершаются аварийной ситуацией. Операции получения, примененные к закрытому каналу, приводят к получению значений, которые были отправлены ранее, до тех пор, пока неполученных значений не останется; любые дальнейшие попытки получить значения приводят к немедленному завершению операции и возврату нулевого значения типа элемента канала.

Чтобы закрыть канал, используется встроенная функция `close`:

```
close(ch)
```

Канал, созданный с помощью простого вызова `make`, называется *небуферизованным* каналом, но `make` принимает необязательный второй аргумент, целое значение, которое называется *емкостью* канала. Если емкость канала ненулевая, `make` создает буферизованный канал.

```
ch = make(chan int)    // Небуферизованный канал
ch = make(chan int, 0) // Небуферизованный канал
ch = make(chan int, 3) // Буферизованный канал с емкостью 3
```

Пока что мы будем рассматривать небуферизованные каналы, а к буферизованным каналам обратимся в разделе 8.4.4.

8.4.1. Небуферизованные каналы

Операция отправления в небуферизованный канал блокирует go-подпрограмму до тех пор, пока другая go-подпрограмма не выполнит соответствующее получение из того же канала, после чего значение становится переданным, и обе go-подпрограммы продолжаются. И наоборот, если первой сделана попытка выполнить операцию получения, принимающая go-подпрограмма блокируется до тех пор, пока другая go-подпрограмма не выполнит отправление значения в тот же канал.

Связь по небуферизованному каналу приводит к *синхронизации* операций отправления и получения. По этой причине небуферизованные каналы иногда называют *синхронными*. Когда значение отправляется в небуферизованный канал, получение значения *предшествует* продолжению работы отправляющей go-подпрограммыПри рассмотрении параллелизма, когда мы говорим, что *x предшествует y*, мы не просто имеем в виду, что *x* происходит по времени раньше, чем *y*; мы подразумеваем, что *x* гарантированно поступает именно так и что все выполненные им предыдущие действия, такие как обновления переменных, завершены и вы можете полностью на них полагаться.

Когда *x* не предшествует *y* и не происходит после *y*, мы говорим, что *x выполняется параллельно с y*. Это не означает, что *x* и *y* обязательно одновременны, просто мы не можем ничего утверждать об их последовательности. Как мы увидим в следующей главе, чтобы избежать проблем, которые возникают, когда две go-подпрограммы одновременно обращаются к одной и той же переменной, определенные события во время выполнения программы необходимо упорядочивать.

Клиентская программа в разделе 8.3 копирует входные данные на сервер в своей главной go-подпрограмме, так что клиентская программа завершается, как только закрывается входной поток, даже если имеется работающая в фоновом режиме go-подпрограмма. Чтобы перед выходом программа ожидала завершения фоновых go-подпрограмм, для синхронизации двух go-подпрограмм мы используем канал:

gopl.io/ch8/netcat3

```
func main() {
    conn, err := net.Dial("tcp", "localhost:8000")
    if err != nil {
        log.Fatal(err)
    }
    done := make(chan struct{})
    go func() {
        io.Copy(os.Stdout, conn) // Примечание: игнорируем ошибки
        log.Println("done")
        done <- struct{}{}       // Сигнал главной go-подпрограмме
    }()
    mustCopy(conn, os.Stdin)
    conn.Close()
    <-done
    // Ожидание завершения фоновой go-подпрограммы
}
```

Когда пользователь закрывает стандартный входной поток, происходит возврат из функции `mustCopy`, и главная go-подпрограмма вызывает `conn.Close()`, закрывая обе половины сетевого подключения. Закрытие записывающей половины соединения заставляет сервер увидеть признак конца файла. Закрытие считывающей половины приводит к тому, что вызов `io.Copy` в go-подпрограмме возвращает ошибку "чтение из закрытого соединения" (поэтому мы убрали запись журнала ошибок); в упражнении 8.3 предлагается лучшее решение. (Обратите внимание, что инструкция `go` вызывает литерал функции — это весьма распространенная конструкция.)

Перед завершением работы фоновая go-подпрограмма заносит в журнал соответствующее сообщение, а затем отправляет значение в канал `done`. Главная go-подпрограмма до тех пор, пока не получит это значение, находится в состоянии ожидания. В результате перед завершением работы программа всегда заносит в журнал сообщение `"done"`.

Сообщения, пересылаемые через каналы, имеют два важных аспекта. Каждое сообщение имеет значение, но иногда важны сам факт передачи сообщения и момент, когда эта передача происходит. Когда мы хотим подчеркнуть этот аспект передачи сообщений, мы называем их *событиями*. Когда событие не несет дополнительной информации, т.е. его единственная цель — это синхронизация, мы подчеркиваем этот факт, используя канал с типом элементов `struct{}`, хотя не менее распространено применение для той же цели каналов с типом значений `bool` или `int`, поскольку выражение `done <- 1` короче выражения `done <- struct{}{}`.

Упражнение 8.3. В программе `netcat3` значение интерфейса `conn` имеет конкретный тип `*net.TCPConn`, который представляет TCP-соединение. TCP-соединение состоит из двух половин, которые могут быть закрыты независимо с использованием методов `CloseRead` и `CloseWrite`. Измените главную go-подпрограмму `netcat3` так, чтобы она закрывала только записывающую половину соединения, так, чтобы программа продолжала выводить последние эхо от сервера `reverb1` даже после того, как стандартный ввод будет закрыт. (Сделать это для сервера `reverb2` труднее; см. упражнение 8.4.)

8.4.2. Конвейеры

Каналы могут использоваться для подключения go-подпрограмм так, чтобы выход одной из них был входом для другой. Это называется *конвейером* (pipeline). Показанная ниже программа состоит из трех go-подпрограмм, соединенных двумя каналами, как схематически показано на рис. 8.1.

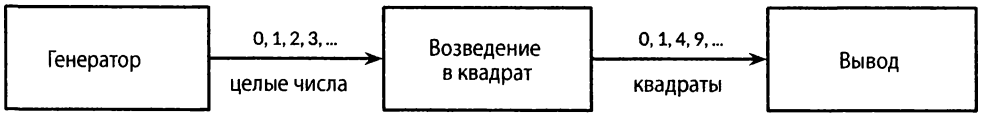


Рис. 8.1. Трехстадийный конвейер

Первая программа генерирует целые числа 0, 1, 2, ... и отправляет их по каналу второй go-подпрограмме, которая получает значения, возводит их в квадрат и передает по следующему каналу третьей go-подпрограмме, которая выводит получаемые значения на экран. Для ясности примера мы намеренно выбрали очень простые функции, хотя, конечно, они слишком тривиальны, чтобы выделять их в отдельные go-подпрограммы в реальной программе:

gopl.io/ch8/pipeline1

```
func main() {
    naturals := make(chan int)
    squares := make(chan int)

    // Генерация
    go func() {
        for x := 0; ; x++ {
            naturals <-x
        }
    }()

    // Возведение в квадрат
    go func() {
        for {
            x := <-naturals
            squares <- x * x
    }
    }()

    // Вывод (в главной go-подпрограмме)
    for {
        fmt.Println(<-squares)
    }
}
```

Как можно ожидать, программа печатает бесконечный ряд квадратов 0, 1, 4, 9 и т.д. Конвейеры, подобные показанному, можно найти в "долгоиграющих" серверах, где каналы используются для непрерывного взаимодействия между go-подпрограммами, содержащими бесконечные циклы. Но что делать, если мы хотим отправить через конвейер только конечное количество значений?

Если отправитель знает, что в канал больше не будут отправляться никакие дальнейшие значения, полезно сообщить этот факт получающей go-подпрограмме, чтобы она могла прекратить ожидание. Это достигается путем *закрытия* канала с помощью встроенной функции `close`:

```
close(naturals)
```

После закрытия канала все последующие операции отправления будут вызывать аварийную ситуацию. После того как закрытый канал *опустошается*, т.е. после получения последнего элемента, все последующие операции получения выполняются без

блокировки, но дают нулевые значения. Закрытие канала `naturals` выше приведет к тому, что go-подпрограмма возведения в квадрат будет нескончаемым потоком получать нулевые значения и направлять их go-подпрограмме вывода.

Не существует способа непосредственно проверить, закрыт ли канал, но есть вариант операции получения, которая возвращает два результата: полученный из канала элемент и логическое значение, условно называемое `ok`, которое равно `true` при успешном получении значения и `false` — при получении из закрытого и опустошенного канала. Используя эту возможность, мы можем изменить цикл в go-подпрограмме возведения в квадрат так, чтобы остановить его, когда канал `naturals` опустошается, и, в свою очередь, закрыть канал `squares`.

```
// Возведение в квадрат
go func() {
    for {
        x, ok := <-naturals
        if !ok {
            break // Канал закрыт и опустошен
        }
        squares <- x * x
    }
    close(squares)
}()
```

Из-за распространенности ситуации и неуклюжести представленного выше синтаксиса язык позволяет использовать для получения всех значений из канала цикл по диапазону. Это более удобный синтаксис для получения всех значений, отправленных в канал, и завершения работы после получения последнего элемента.

В показанном ниже конвейере go-подпрограмма генерации заканчивает свой цикл после отправления ста элементов и закрывает канал `naturals`. Это заставляет go-подпрограмму возведения в квадрат закончить свой цикл и закрыть канал `squares`. (В более сложной программе для функций генерации и возведения в квадрат может иметь смысл в самом начале этих функций отложить вызов `close` с помощью ключевого слова `defer`.) Наконец главная go-подпрограмма завершает свой цикл и программа завершает работу.

gopl.io/ch8/pipeline2

```
func main() {
    naturals := make(chan int)
    squares := make(chan int)

    // Генерация
    go func() {
        for x := 0; x < 100; x++ {
            naturals <- x
        }
        close(naturals)
    }()
```

```
    // Возведение в квадрат
    go func() {
        for x := range naturals {
            squares <- x * x
        }
        close(squares)
    }()

    // Вывод (в главной go-подпрограмме)
    for x := range squares {
        fmt.Println(x)
    }
}
```

Вам не нужно закрывать каждый канал по завершении работы с ним. Необходимо закрывать каналы тогда, когда важно сообщить принимающей go-подпрограмме, что все данные уже отправлены. Ресурсы канала, который сборщик мусора определяет как недоступный, будут освобождены в любом случае, независимо от того, закрыт ли он. (Не путайте это с операцией закрытия открытых файлов; вызывать метод `Close` *важно* для каждого файла, работа с которым завершена.)

Попытка закрыть уже закрытый канал вызывает аварийную ситуацию, так же как и закрытие нулевого канала. Закрытие каналов имеет и другое применение — в качестве механизма оповещения, который мы рассмотрим в разделе 8.9.

8.4.3. Однонаправленные каналы

По мере роста программ крупные функции естественным образом распадаются на более мелкие фрагменты. В нашем предыдущем примере использованы три go-подпрограммы, соединяющиеся с помощью двух каналов, которые представляют собой локальные переменные функции `main`. Программа естественным образом делится на три функции:

```
func counter(out chan int)
func squarer(out, in chan int)
func printer(in chan int)
```

Функция `squarer`, находящаяся в средине конвейера, получает два параметра — входной и выходной каналы. Оба они имеют один и тот же тип, но используются в разных целях: `in` работает только на получение, а `out` — только на отправление. Имена `in` и `out` выражают это предназначение, но, тем не менее, ничто не мешает функции `squarer` отправлять данные в поток `in` или получать их из потока `out`.

Эта ситуация достаточно типична. Когда канал передается в качестве параметра функции, это почти всегда делается с тем намерением, что он должен использоваться исключительно для отправления или исключительно для получения данных.

Для документирования этого намерения и предотвращения неверного применения система типов Go предоставляет тип *однонаправленного* канала, который обеспечивает только одну операцию — отправление или получение. Тип `chan<-int` представ-

ляет собой канал `int`, предназначенный *только для отправления*, который позволяет отправлять данные, но не получать их. Напротив, тип `<-chan int`, канал `int`, предназначенный *только для получения*, позволяет получать, но не отправлять данные. (Положение стрелки `<-` относительно ключевого слова `chan` является мнемоническим.) Нарушения применения таких каналов обнаруживаются во время компиляции.

Поскольку операция `close` утверждает, что в канал больше не будет отправления данных, вызывать ее может только отправляющая go-подпрограмма; по этой причине попытка закрыть канал только для получения приводит к ошибке времени компиляции.

Давайте еще раз рассмотрим конвейер, возводящий числа в квадрат, на этот раз — с применением однонаправленных каналов:

gopl.io/ch8/pipeline3

```
func counter(out chan<- int) {
    for x := 0; x < 100; x++ {
        out <- x
    }
    close(out)
}

func squarer(out chan<- int, in <-chan int) {
    for v := range in {
        out <- v * v
    }
    close(out)
}

func printer(in <- chan int) {
    for v := range in {
        fmt.Println(v)
    }
}

func main() {
    naturals := make(chan int)
    squares := make(chan int)

    go counter(naturals)
    go squarer(squares, naturals)
    printer(squares)
}
```

Вызов `counter(naturals)` неявно преобразует `naturals`, значение типа `chan int`, в тип параметра, `chan<-int`. Вызов `printer(squares)` выполняет аналогичное неявное преобразование в `<-chan int`. Преобразования двунаправленных каналов в однонаправленные разрешается в любом присваивании. Однако обратное не верно: если у вас есть значение однонаправленного типа, такого как `chan<-int`, то

нет способа получить из него значение типа `chan int`, которое ссылается на структуру данных того же канала.

8.4.4. Буферизованные каналы

Буферизованный канал имеет очередь элементов. Максимальный размер очереди определяется при создании канала с помощью аргумента емкости функции `make`. Приведенная ниже инструкция создает буферизованный канал, способный хранить три строковых значения. На рис. 8.2 графически представлены `ch` и канал, на который он указывает:

```
ch = make(chan string, 3)
```

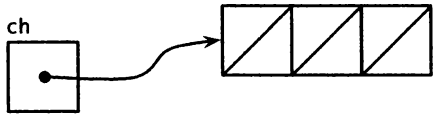


Рис. 8.2. Пустой буферизованный канал

Операция отправления в буферизованный канал вставляет отправляемый элемент в конец очереди, а операция получения удаляет первый элемент из очереди. Если канал заполнен, операция отправления блокирует свою go-подпрограмму до тех пор, пока другая go-подпрограмма не освободит место, получив данные из канала. И наоборот, если канал пуст, операция получения блокирует go-подпрограмму до того момента, пока в канал не будет послано значение из другой go-подпрограммы.

В указанный канал можно отправить до трех значений без блокировки go-подпрограммы:

```
ch <-"A"
ch <-"B"
ch <-"C"
```

В этот момент канал оказывается заполненным (рис. 8.3), и инструкция отправления четвертого элемента приводит к блокировке go-подпрограммы.

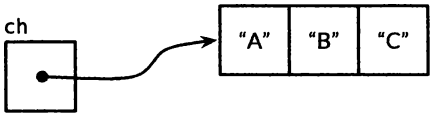


Рис. 8.3. Заполненный буферизованный канал

Если мы получим из канала одно значение

```
fmt.Println(<-ch) // "A"
```

то канал станет ни пустым, ни заполненным (рис. 8.4), так что как операция отправления, так и операция получения будут выполнены без блокировки. Таким образом, буфер канала разделяет go-подпрограммы отправления и получения.

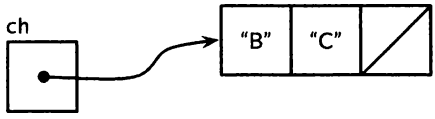


Рис. 8.4. Частично заполненный буферизованный канал

В маловероятном случае, когда программе необходимо знать емкость буфера канала, ее можно получить, вызвав встроенную функцию `cap`:

```
fmt.Println(cap(ch)) // "3"
```

При применении к каналу встроенная функция `len` возвращает количество элементов, находящихся в настоящее время в буфере. Поскольку в параллельной программе эта информация, скорее всего, окажется устаревшей сразу по получении, ее применение ограничено, но она может оказаться полезной для оптимизации производительности или при отладке.

```
fmt.Println(len(ch)) // "2"
```

После еще двух операций получения канал снова станет пустым, а четвертая операция заблокирует работу go-подпрограммы:

```
fmt.Println(<-ch) // "B"
fmt.Println(<-ch) // "C"
```

В этом примере операции отправления и получения выполнялись одной и той же go-подпрограммой, но в реальных программах они обычно выполняются различными go-подпрограммами. Новички иногда соблазняются простым синтаксисом и используют буферизованные каналы в пределах одной go-подпрограммы в качестве очереди, но это ошибка. Каналы глубоко связаны с планированием go-подпрограмм, и без другой go-подпрограммы, получающей данные из канала, отправитель — да, пожалуй, вся программа в целом — рискует быть навсегда заблокированным. Если все, что вам нужно, — это простая очередь, воспользуйтесь срезом.

В приведенном ниже примере показано приложение буферизованного канала. Оно выполняет параллельные запросы к трем *зеркалам*, т.е. к эквивалентным, но географически разнесенным серверам. Их ответы отправляются в буферизованный канал, а затем оттуда поступает только самый быстрый первый ответ. Таким образом, `mirroredQuery` возвращает результат даже до того, как остальные медленные серверы установят соединение. (Кстати, для нескольких go-подпрограмм вполне нормально передавать значения в один и тот же канал одновременно, как в приведенном примере, как и получать данные из одного и того же канала.)

```
func mirroredQuery() string {
    responses := make(chan string, 3)
    go func() { responses <-request("asia.gopl.io") }()
    go func() { responses <-request("europe.gopl.io") }()
    go func() { responses <-request("americas.gopl.io") }()
```

```
    return <-responses  // Возврат самого быстрого ответа
}
func request(hostname string) (response string) { /* ... */ }
```

Если бы мы использовали канал без буферизации, две более медленные go-подпрограммы были бы заблокированы при попытках отправить свои ответы в канал, из которого никто никогда не пытался бы считывать данные. Эта ситуация, которая называется *утечкой go-подпрограмм*, была бы ошибкой. В отличие от переменных, потерянные go-подпрограммы не собираются сборщиком мусора автоматически, поэтому важно гарантировать, что go-подпрограммы должны прекратиться сами, когда они больше не нужны.

Выбор между буферизованными и небуферизованными каналами, как и выбор емкости буферизованных каналов, может влиять на корректность работы программы. Небуферизованные каналы дают более надежные гарантии синхронизации, потому что каждая операция отправления синхронизируется с соответствующей операцией получения; в случае буферизованных каналов эти операции разделены. Кроме того, когда мы знаем верхнюю границу количества значений, отправляемых в канал, нет ничего необычного в создании буферизованного канала данного размера с выполнением всех отправлений еще до получения первого значения из канала. Неспособность выделить достаточную емкость буфера может привести ко взаимоблокировке программы.

Буферизация канала может повлиять и на производительность программы. Представьте себе трех кондитеров, из которых один печет торт, второй поливает его глазурью, а третий разукрашивает торт перед передачей на продажу. В кухне, где мало места, каждый кондитер, который закончил свою операцию с тортом, вынужден дожидаться, когда следующий кондитер будет готов принять его работу. Это аналог канала связи без буферизации.

Если же между кондитерами есть место для одного торта, то кондитер может поставить там готовый торт и браться за следующий. Это аналог канала с буфером емкостью 1. Если кондитеры работают в среднем примерно одинаково, большинство тортов пройдут быстро: временные различия в работе будут сглажены наличием буфера. Чем больше места между поварами — т.е. чем больше буфера, — тем более сильные вариации будут сглажены без приостановки конвейера (как, например, если один повар вынужден ненадолго отлучиться, после чего наверстывает упущенное).

С другой стороны, если на более ранней стадии конвейера работа идет быстрее, чем на следующем этапе, буфер между ними будет б*о*льшую часть времени заполнен. И наоборот, если на последующей стадии работа идет быстрее, буфер в основном пусты. В этом случае буфер не приносит никакой пользы.

Метафора такого конвейера полезна для каналов и go-подпрограмм. Например, если работа на втором этапе является более сложной, то один кондитер может не иметь возможности обеспечить необходимую скорость, так что мы могли бы нанять еще одного кондитера ему в помощь — для выполнения той же задачи, но работающего самостоятельно. Это аналогично созданию еще одной go-подпрограммы, сообщающейся с другими через те же каналы.

Здесь недостаточно места, чтобы привести полный исходный текст, но пакет `gopl.io/ch8/cake` имитирует этот магазин тортов, с несколькими параметрами, которые можно изменять. Он включает в себя функции производительности (раздел 11.4) для нескольких сценариев, описанных выше.

8.5. Параллельные циклы

В этом разделе мы рассмотрим некоторые распространенные шаблоны для параллельного выполнения всех итераций цикла. Мы будем рассматривать задачу создания малых эскизов для набора полномасштабных изображений. Пакет `gopl.io/ch8/thumbnail` содержит функцию `ImageFile`, которая масштабирует одно изображение. Мы не будем показывать ее реализацию, она может быть загружена с сайта `gopl.io`.

gopl.io/ch8/thumbnail

```
package thumbnail

// ImageFile считывает изображение из infile и
// записывает эскиз в тот же каталог.
// Возвращает сгенерированное имя файла, например "foo.thumb.jpg".
func ImageFile(infile string) (string, error)
```

Приведенная ниже программа проходит по списку имен файлов изображений и генерирует для каждого миниатюрный эскиз:

gopl.io/ch8/thumbnail

```
// makeThumbnails создает эскизы для файлов из списка.
func makeThumbnails(filenames []string) {
    for _, f := range filenames {
        if _, err := thumbnail.ImageFile(f); err != nil {
            log.Println(err)
        }
    }
}
```

Очевидно, что порядок, в котором обрабатываются файлы, не имеет значения, поскольку каждая операция масштабирования является независимой от всех остальных. Подобные задачи, которые состоят исключительно из полностью независимых одна от другой подзадач, описываются как *ошеломляюще параллельные* (embarrassingly parallel). Для ошеломляюще параллельных задач проще всего реализовать параллельное выполнение и получить производительность, которая линейно растет с увеличением степени параллелизма.

Давайте выполним все эти операции параллельно, тем самым скрывая задержку файлового ввода-вывода и используя несколько процессоров для масштабирования изображений. Наша первая попытка получить параллельную версию просто добавляет ключевое слово `go`. Пока что мы будем игнорировать все возможные ошибки.

```
// Примечание: неправильно!
func makeThumbnails2(filenames []string) {
    for _, f := range filenames {
        go thumbnail.ImageFile(f) // Примечание: игнорируем ошибки
    }
}
```

Эта версия работает очень быстро, на самом деле даже слишком быстро, так как занимает куда меньше времени, чем оригинал, даже когда срез имен файлов содержит только один элемент. Если нет параллелизма, то как же параллельная версия может работать быстрее? Ответ прост — возврат из `makeThumbnails` выполняется до того, как выполняется все то, что эта функция должна делать. Он запускает go-подпрограммы, по одной для каждого имени файла, но не ждет их завершения.

Увы, не существует непосредственного способа дождаться завершения go-подпрограммы, но мы можем изменить внутреннюю go-подпрограмму таким образом, чтобы она сообщала о своем завершении внешней go-подпрограмме, отправляя событие в общий канал. Поскольку мы знаем, что имеется ровно `len(filenames)` внутренних go-подпрограмм, внешней программе нужно просто подсчитать нужное количество событий, прежде чем завершиться:

```
// makeThumbnails3 параллельно делает эскизы определенных файлов.
func makeThumbnails3(filenames []string) {
    ch := make(chan struct{})
    for _, f := range filenames {
        go func(f string) {
            thumbnail.ImageFile(f) // Примечание: игнорируем ошибки
            ch <-struct{}{}
        }(f)
    }
    // Ожидаем завершения go-подпрограмм
    for range filenames {
        <-ch
    }
}
```

Обратите внимание, что мы передали значение `f` как явный аргумент литеральной функции вместо того, чтобы использовать объявление `f` из охватывающего цикла `for`:

```
for _, f := range filenames {
    go func() {
        thumbnail.ImageFile(f) // Примечание: неправильно!
        // ...
    }()
}
```

Вспомните о проблеме захвата переменной цикла внутри анонимной функции, описанной в разделе 5.6.1. Выше одна переменная `f` разделяется всеми значениями — анонимными функциями и обновляется каждой очередной итерацией. Ко времени,

когда новая go-подпрограмма начинает выполнение литеральной функции, цикл, возможно, уже обновил `f` и начал другую итерацию или (что даже более вероятно) завершился полностью, так что, когда эти go-подпрограммы добрались до чтения значения `f`, все они увидели значение для последнего элемента среза. Используя явный параметр, мы гарантируем, что применяется значение `f`, которое является текущим в момент выполнения инструкции `go`.

Что делать, если мы хотим возвращать значения от каждой рабочей go-подпрограммы в главную? Если вызову `thumbnail.ImageFile` не удается создать файл, он возвращает сообщение об ошибке. Следующая версия `makeThumbnails` возвращает первую ошибку, которую она получает от любой из операций масштабирования:

```
// makeThumbnails4 параллельно делает эскизы определенных файлов.
// Возвращает ошибку при сбое на любом шаге.
func makeThumbnails4(filenames []string) error {
    errors := make(chan error)

    for _, f := range filenames {
        go func(f string) {
            _, err := thumbnail.ImageFile(f)
            errors <- err
        }(f)
    }

    for range filenames {
        if err := <-errors; err != nil {
            return err // Примечание: неверно: утечка go-подпрограмм!
        }
    }
    return nil
}
```

В этой функции содержится тонкая ошибка. Встретив первую ненулевую ошибку, она возвращает ее вызывающей функции, не позволяя go-подпрограмме опустошить канал `errors`. В результате каждая оставшаяся работающая go-подпрограмма будет навсегда заблокирована, если попытается отправить значение в этот канал, и никогда не завершится. Эта ситуация, утечка go-подпрограмм (раздел 8.4.4), может привести к остановке всей программы или нехватке памяти.

Самым простым решением является использование буферизованного канала с достаточной емкостью, который не будет блокировать рабочие go-подпрограммы при отправке сообщений. (Альтернативное решение заключается в создании еще одной go-подпрограммы для опустошения канала, в то время как основная go-подпрограмма без промедления возвращает первую ошибку.)

Очередная версия `makeThumbnails` использует буферизованный канал для возврата имен сгенерированных файлов вместе со всеми ошибками.

```
// makeThumbnails5 параллельно делает эскизы определенных файлов.
// Возвращает имена сгенерированных файлов в произвольном порядке
```

```
// или ошибку при сбое на любом шаге.
func makeThumbnails5(filenames []string)
                    (thumbfiles []string, err error){
    type item struct {
        thumbfile string
        err       error
    }

    ch := make(chan item, len(filenames))
    for _, f := range filenames {
        go func(f string) {
            var it item
            it.thumbfile, it.err = thumbnail.ImageFile(f)
            ch <- it
        }(f)
    }

    for range filenames {
        it := <- ch
        if it.err != nil {
            return nil, it.err
        }
        thumbfiles = append(thumbfiles, it.thumbfile)
    }

    return thumbfiles, nil
}
```

Наша окончательная версия `makeThumbnails`, показанная ниже, возвращает общее количество байтов, занятых новыми файлами. Однако в отличие от предыдущих версий, она получает имена файлов не как срез, а по каналу строк, поэтому мы не можем предсказать количество итераций цикла.

Чтобы узнать, когда последняя go-подпрограмма закончит работу (эта go-подпрограмма может не быть последней из запущенных на выполнение), нужно увеличивать счетчик перед запуском каждой go-подпрограммы и уменьшать его после завершения. Для этого требуется счетчик особого рода, с которым могут безопасно работать несколько go-подпрограмм и который предоставляет возможность ожидания, пока он не станет равным нулю. Этот тип счетчика известен как `sync.WaitGroup`, и приведенный ниже код показывает, как его использовать:

```
// makeThumbnails6 параллельно делает эскизы определенных файлов.
// Возвращает общее количество байтов в созданных файлах.
func makeThumbnails6(filenames <-chan string) int64 {
    sizes := make(chan int64)
    var wg sync.WaitGroup // Количество работающих go-подпрограмм
    for f := range filenames {
        wg.Add(1)
        // Рабочая go-подпрограмма
```

```
        go func(f string) {
            defer wg.Done()
            thumb, err := thumbnail.ImageFile(f)
            if err != nil {
                log.Println(err)
                return
            }
            info, _ := os.Stat(thumb) // Игнорируем ошибки
                sizes <- info.Size()
        }(f)
    }

    // Ожидание счетчика
    go func() {
        wg.Wait()
        close(sizes)
    }()

    var total int64
    for size := range sizes {
        total += size
    }
    return total
}
```

Обратите внимание на асимметрию методов `Add` и `Done`. Метод `Add`, который увеличивает значение счетчика, должен быть вызван перед началом рабочей go-подпрограммы, но не внутри него; в противном случае вы не можете быть уверены, что `Add` *предшествует* go-подпрограмме ожидания счетчика, которая вызывает метод `Wait`. Кроме того, `Add` принимает параметр, а `Done` — нет; вызов последнего метода эквивалентен вызову `Add(-1)`. Мы использовали `defer`, чтобы гарантировать, что значение счетчика уменьшается даже в случае ошибки. Показанная структура кода является распространенным идиоматическим шаблоном для параллельно выполняемых циклов, когда мы не знаем количества итераций заранее.

Канал `sizes` передает размер каждого файла обратно в главную go-подпрограмму, которая получает эти значения с помощью цикла по диапазону и вычисляет сумму. Обратите внимание, как мы создаем go-подпрограмму ожидания счетчика, которая ждет завершения всех рабочих go-подпрограмм. Эти две операции, ожидания счетчика и закрытия канала, должны выполняться параллельно с циклом, работающим с каналом `sizes`. Рассмотрим альтернативы: если операция ожидания находится в главной go-подпрограмме перед циклом, она никогда не завершится, а если ее поместить после цикла, она окажется недоступной, поскольку ничто не закроет канал, а значит, цикл никогда не прекратится.

На рис. 8.5 показана последовательность событий в функции `makeThumbnails6`. Вертикальные линии представляют go-подпрограммы. Тонкие отрезки указывают периоды ожидания, толстые — деятельности. Наклонные стрелки указывают события, которые синхронизируют одну go-подпрограмму с другой. Ось времени на-

правлена вниз. Обратите внимание, как главная go-подпрограмма проводит бо́льшую часть своего времени в цикле по диапазону в состоянии ожидания, пока рабочая go-подпрограмма не отправит значение, или go-подпрограмма ожидания счетчика не закроет канал.

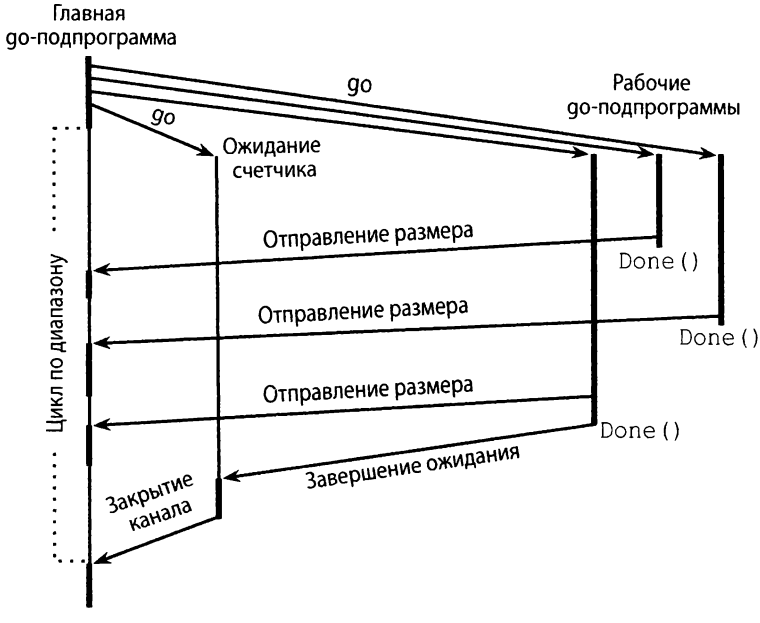


Рис. 8.5. Последовательность событий в `makeThumbnails6`

Упражнение 8.4. Модифицируйте сервер `reverb2` так, чтобы он использовал по одному объекту `sync.WaitGroup` для каждого соединения для подсчета количества активных go-подпрограмм `echo`. Когда он обнуляется, закрывайте пишущую половину TCP-соединения, как описано в упражнении 8.3. Убедитесь, что вы изменили клиентскую программу `netcat3` из этого упражнения так, чтобы она ожидала последние ответы от параллельных go-подпрограмм сервера даже после закрытия стандартного ввода.

Упражнение 8.5. Возьмите существующую последовательную программу, такую как программа вычисления множества Мандельброта из раздела 3.3 или вычисления трехмерной поверхности из раздела 3.2, и выполните ее главный цикл параллельно, с использованием каналов. Насколько быстрее стала работать программа на многопроцессорной машине? Каково оптимальное количество используемых go-подпрограмм?

8.6. Пример: параллельный веб-сканер

В разделе 5.6 мы создали простой веб-сканер, который исследовал граф ссылок в Интернете в порядке поиска в ширину. В этом разделе мы сделаем его параллельным, чтобы независимые вызовы `crawl` могли использовать параллелизм операций

ввода-вывода в Интернете. Функция `crawl` остается той же, что и в `gopl.io/ch5/findlinks3`:

gopl.io/ch8/crawl1
```
func crawl(url string) []string {
    fmt.Println(url)
    list, err := links.Extract(url)
    if err != nil {
        log.Print(err)
    }
    return list
}
```

Функция `main` напоминает `breadthFirst` (раздел 5.6). Как и ранее, `worklist` записывает очередь элементов, требующих обработки, каждый элемент списка URL сканируется, но на этот раз вместо представления очереди в виде среза мы используем канал. Каждый вызов `crawl` осуществляется в своей go-подпрограмме и передает обнаруженные ссылки обратно в список.

```
func main() {
    worklist := make(chan []string)

    // Запуск с аргументами командной строки.
    go func() { worklist <- os.Args[1:] }()

    // Параллельное сканирование.
    seen := make(map[string]bool)
    for list := range worklist {
        for _, link := range list {
            if !seen[link] {
                seen[link] = true
                go func(link string) {
                    worklist <- crawl(link)
                }(link)
            }
        }
    }
}
```

Обратите внимание, что go-подпрограмма сканирования принимает `link` в качестве явного параметра, чтобы избежать проблемы захвата переменной цикла, которую мы впервые увидели в разделе 5.6.1. Обратите также внимание, что первоначальные аргументы командной строки должны передаваться в рабочий список в отдельной go-подпрограмме, чтобы избежать *взаимоблокировки*, ситуации "зависания", в которой основная go-подпрограмма и go-подпрограммы сканирования пытаются отправлять информацию друг другу в то время, когда ни одна из них ее не получает. Альтернативное решение заключается в использовании буферизованного канала.

Теперь сканер обладает высокой параллельностью и выводит поток URL, но в нем имеются две проблемы. Первая проблема проявляется в виде сообщений об ошибках в журнале после нескольких секунд работы:

```
$ go build gopl.io/ch8/crawl1
$ ./crawl1 http://gopl.io/
http://gopl.io/
https://golang.org/help/
https://golang.org/doc/
https://golang.org/blog/
...
2015/07/15 18:22:12 Get ...: dial tcp: lookup blog.golang.org:
                                нет такого хоста
2015/07/15 18:22:12 Get ...: dial tcp 23.21.222.120:443: socket:
                                слишком много открытых файлов
...
```

Первое сообщение — это странное уведомление об ошибке поиска DNS для существующего домена. Следующее сообщение об ошибке поясняет причины: программа создает так много сетевых подключений сразу, что превышает предел количества одновременно открытых файлов, установленный для каждого процесса, что приводит к сбоям таких операций, как DNS-запросы и вызовы `net.Dial`.

Эта программа оказывается *слишком* параллельной. Неограниченный параллелизм редко является хорошей идеей, поскольку в системе всегда есть ограничивающие факторы, такие как количество ядер процессора для вычислительной нагрузки, количество шпинделей и головок для локальных дисковых операций ввода-вывода, пропускной способности сети для потоковой загрузки. Решение заключается в том, чтобы ограничить количество параллельных обращений к ресурсам в соответствии с доступным уровнем параллелизма. Простейший способ добиться этого в нашем примере — обеспечить не более чем n вызовов `links.Extract` одновременно, где n — значение, меньшее, чем ограничение на количество открытых файлов (скажем, 20). Это аналогично тому, как швейцар в переполненном ночном клубе впускает гостя только тогда, когда уйдут некоторые другие гости.

Мы можем ограничить параллелизм с помощью буферизованного канала с емкостью n для моделирования примитива параллелизма, который называется *подсчитывающим семафором*. Концептуально каждый из n свободных слотов в буфере канала представляет маркер. Отправление значения в канал захватывает маркер, а получение значения из канала его освобождает, создавая новый свободный слот. Это гарантирует, что без промежуточного получения значения из канала в него может быть отправлено не более n значений. (Хотя более наглядным может быть рассмотрение как маркеров *заполненных* слотов в буфере канала, но использование свободных слотов позволяет избежать необходимости заполнять буфер канала после его создания). Так как тип элемента канала не важен, мы будем использовать `struct{}`, имеющий нулевой размер.

Давайте перепишем функцию `crawl` так, чтобы вызов `links.Extract` был помещен между операциями захвата и освобождения маркера, тем самым гарантируя,

что одновременно активными являются не более 20 вызовов. Хорошая практика заключается в том, чтобы операции с семафорами располагались как можно ближе к операциям ввода-вывода, которые они регулируют.

gopl.io/ch8/crawl2

```
// tokens представляет собой подсчитывающий семафор, используемый
// для ограничения количества параллельных запросов величиной 20.
var tokens = make(chan struct{}, 20)
func crawl(url string) []string {
    fmt.Println(url)
    tokens <- struct{}{}  // Захват маркера
    list, err := links.Extract(url)
    <- tokens             // Освобождение маркера
    if err != nil {
        log.Print(err)
    }
    return list
}
```

Вторая проблема заключается в том, что программа никогда не завершается, даже после обнаружения всех достижимых из начального URL ссылок. (Конечно, вы вряд ли заметите эту проблему, если только не выберете тщательно первоначальный URL или реализуете возможность ограничения глубины из упражнения 8.6.) Для завершения программы мы должны выйти из основного цикла, когда список пуст *и* нет активных сканирующих go-подпрограмм.

```
func main() {
    worklist := make(chan []string)
    var n int // Количество ожидающих отправки в список

    // Запуск с аргументами командной строки.
    n++
    go func() { worklist <- os.Args[1:] }()

    // Параллельное скаирование веб.
    seen := make(map[string]bool)
    for ; n > 0; n-- {
        list := <- worklist
        for _, link := range list {
            if !seen[link] {
                seen[link] = true
                n++
                go func(link string) {
                    worklist <- crawl(link)
                }(link)
            }
        }
    }
}
```

В этой версии счетчик n отслеживает количество отправок в рабочий список, которые находятся в состоянии ожидания. Каждый раз, зная, что элемент должен быть отправлен в рабочий список, мы увеличиваем n, один раз перед тем, как отправлять в список первоначальные аргументы командной строки, и по разу перед каждым запуском go-подпрограммы сканирования. Главный цикл завершается, когда n уменьшается до нуля, поскольку при этом больше нет никакой работы, которую предстоит сделать.

Теперь параллельный сканер работает (без ошибок) примерно в 20 раз быстрее сканера из раздела 5.6 и корректно завершается по окончании работы.

В приведенной ниже программе показано альтернативное решение проблемы чрезмерного параллелизма. Эта версия использует исходную функцию `crawl` без подсчитывающего семафора, но вызывает ее из одной из 20 долгоживущих go-подпрограмм сканирования, гарантируя тем самым, что одновременно активны не более 20 HTTP-запросов.

gopl.io/ch8/crawl3

```
func main() {
    worklist := make(chan []string)  // Список URL, могут быть дубли.
    unseenLinks := make(chan string) // Удаление дублей.

    // Добавление в список аргументов командной строки.
    go func() { worklist <- os.Args[1:] }()

    // Создание 20 go-подпрограмм сканирования для выборки
    // всех непросмотренных ссылок.
    for i := 0; i < 20; i++ {
        go func() {
            for link := range unseenLinks {
                foundLinks := crawl(link)
                go func() { worklist <- foundLinks }()
            }
        }()
    }

    // Главная go-подпрограмма удаляет дубликаты из списка
    // и отправляет непросмотренные ссылки сканерам.
    seen := make(map[string]bool)
    for list := range worklist {
        for _, link := range list {
            if !seen[link] {
                seen[link] = true
                unseenLinks <- link
            }
        }
    }
}
```

Go-подпрограммы сканеров используют один и тот же канал, `unseenLinks`. Главная go-подпрограмма отвечает за исключение дублирования элементов, которые она получает из рабочего списка, а затем отправляет каждую непросмотренную ссылку по каналу `unseenLinks` go-подпрограмме сканирования.

Отображение `seen` *замкнуто* в пределах главной go-подпрограммы, т.е. доступ к нему имеет только главная go-подпрограмма. Подобно другим разновидностям сокрытия информации, замкнутость помогает нам обосновать корректность программы. Например, локальные переменные не могут упоминаться по имени за пределами функции, в которой они были объявлены; переменные, не сбежавшие (раздел 2.3.4) из функции, недоступны вне этой функции; а инкапсулированные поля объекта не могут быть доступны ничему, кроме методов этого объекта. Во всех случаях сокрытие информации помогает ограничить нежелательные взаимодействия между частями программы.

Ссылки, найденные функцией `crawl`, отправляются в рабочий список из выделенной go-подпрограммы, чтобы избежать взаимоблокировки.

Для экономии места мы не рассматривали в этом примере проблему завершения программы.

Упражнение 8.6. Добавьте ограничение по глубине в параллельный сканер. Иначе говоря, если пользователь устанавливает `-depth=3`, то выбираются только те URL, которые достижимы через цепочку не более чем из трех ссылок.

Упражнение 8.7. Напишите параллельную программу, которая создает локальное зеркало веб-сайта, загружая все доступные страницы и записывая их в каталог на локальном диске. Выбираться должны только страницы в пределах исходного домена (например, `golang.org`). URL в страницах зеркала должны при необходимости быть изменены таким образом, чтобы они ссылались на зеркальную страницу, а не на оригинал.

8.7. Мультиплексирование с помощью `select`

Приведенная ниже программа осуществляет обратный отсчет для запуска ракеты. Функция `time.Tick` возвращает канал, по которому она периодически отправляет события, действуя как метроном. Каждое событие представляет собой значение момента времени, но оно не так интересно, как сам факт его доставки.

gopl.io/ch8/countdown1

```
func main() {
    fmt.Println("Начинаю отсчет.")
    tick := time.Tick(1 * time.Second)
    for countdown := 10; countdown > 0; countdown-- {
        fmt.Println(countdown)
        <- tick
    }
    launch()
}
```

Давайте теперь добавим возможность прервать последовательность запуска, нажав клавишу <Enter> во время обратного отсчета. Сначала мы запустим go-подпрограмму, которая пытается прочитать один байт из стандартного ввода и, если это удастся, отправляет значение в канал, который называется `abort`.

gopl.io/ch8/countdown2

```
abort := make(chan struct{})
go func() {
    os.Stdin.Read(make([]byte, 1)) // Чтение одного байта
    abort <- struct{}{}
}()
```

Теперь каждая итерация цикла обратного отсчета должна ожидать события от одного из двух каналов: канала `tick`, если все в порядке, или канала `abort`, если следует отменить запуск. Мы не можем просто получать значения от каждого канала, потому что первая же операция будет блокирована до ее завершения. Мы должны *мультиплексировать* эти операции, а чтобы это сделать, нужна инструкция `select`:

```
select {
case <-ch1:
    // ...
case x := <-ch2:
    // ...используем x...
case ch3 <-y:
    // ...
default:
    // ...
}
```

Общий вид инструкции `select` показан выше. Как и инструкция `switch`, она имеет ряд вариантов `case` и дополнительный вариант по умолчанию `default`. Каждый вариант определяет *связь* (операции отправления или получения на некоторых каналах) и блок инструкций. Выражение получения может быть как само по себе, как показано в первом случае, так и в коротком объявлении переменной, как показано во втором случае. Вторая форма позволяет ссылаться на полученное значение.

Инструкция `select` ожидает, когда связь для некоторого варианта будет готова к обработке. Затем она выполняет операцию связи и выполняет инструкции из блока для данного варианта. Операции прочих связей не происходят. Инструкция `select` без вариантов, `select{}`, ожидает вечно.

Вернемся к нашей программе запуска ракеты. Функция `time.After` немедленно возвращает канал и запускает новую go-подпрограмму, которая отправляет одно значение в канал по истечении указанного времени. Инструкция `select` ниже ожидает до тех пор, пока не произойдет первое из двух событий, либо событие прерывания запуска, либо событие, указывающее, что прошло 10 секунд. Если прошло 10 секунд без события прерывания, программа выполняет запуск.

```
func main() {
    // ... создание канала abort ...
```

```
    fmt.Println("Начинаю отсчет. Нажмите <Enter> для отказа.")
    select {
    case <- time.After(10 * time.Second):
        // Ничего не делаем.
    case <-abort:
        fmt.Println("Запуск отменен!")
        return
    }
    launch()
}
```

Пример ниже — более тонкий. Канал `ch` с буфером размером 1 поочередно становится то полным, то пустым, так что обрабатывается только один из вариантов: либо отправление при четном `i`, либо получение при `i` нечетном. На экран всегда выводится `0 2 4 6 8`:

```
ch := make(chan int, 1)
for i := 0; i < 10; i++ {
    select {
    case x := <-ch:
        fmt.Println(x) // "0" "2" "4" "6" "8"
    case ch <-i:
    }
}
```

Если готовы несколько вариантов, `select` выбирает один из них случайным образом; это гарантирует, что каждый канал имеет равные шансы быть выбранным. Увеличение размера буфера в предыдущем примере делает его вывод недетерминированным, потому что, когда буфер ни полный, ни пустой, инструкция `select`, образно говоря, бросает монетку.

Давайте сделаем нашу программу запуска выводящей обратный отсчет. Инструкция `select` ниже приводит к тому, что на каждой итерации цикла выполняется ожидание сигнала прерывания в течение секунды, но не дольше.

gopl.io/ch8/countdown3

```
func main() {
    // ... создание канала abort ...
    fmt.Println("Начинаю отсчет. Нажмите <Enter> для отказа.")
    tick := time.Tick(1 * time.Second)
    for countdown := 10; countdown > 0; countdown-- {
        fmt.Println(countdown)
        select {
        case <-tick:
            // Ничего не делаем.
        case <-abort:
            fmt.Println("Запуск отменен!")
            return
        }
    }
```

```
    launch()
}
```

Функция `time.Tick` ведет себя так, как будто создает go-подпрограмму, которая вызывает `time.Sleep` в цикле, отправляя событие всякий раз, когда она "просыпается". Когда функция отсчета завершается, она перестает получать события от `tick`, но соответствующая go-подпрограмма остается работать и пытается отправлять события в канал, из которого их не получает никакая go-подпрограмма, т.е. происходит *утечка go-подпрограммы* (раздел 8.4.4).

Функция `Tick` удобна, но она подходит только тогда, когда отсчеты времени необходимы на протяжении всего жизненного цикла приложения. В противном случае необходимо использовать следующий шаблон:

```
ticker := time.NewTicker(1 * time.Second)

<-ticker.C    // Получение из канала ticker

ticker.Stop() // Остановка go-подпрограммы ticker
```

Иногда мы хотим попытаться отправить данные в канал или получить их из него, но избежать блокировки, если канал не готов, т.е. обеспечить *неблокирующую* связь. Инструкция `select` может сделать и это. Она может иметь вариант по умолчанию `default`, который указывает, что делать, когда ни одно из других сообщений не может быть обработано немедленно.

Инструкция `select` ниже получает значение из канала `abort`, если таковое доступно для получения; в противном случае она ничего не делает. Это высокопроизводительная неблокирующая операция получения; ее многократное повторение называют *опросом* канала.

```
select {
case <-abort:
    fmt.Printf("Запуск отменен!\n")
    return
default:
    // Ничего не делать
}
```

Нулевым значением для канала является `nil`. Возможно, это удивительно, но иногда нулевые каналы полезны. Поскольку операции отправления и получения с нулевым каналом блокируются навсегда, варианты с нулевыми каналами в инструкции `select` никогда не выбираются. Это позволяет нам использовать `nil` для включения или отключения вариантов, которые соответствуют таким возможностям, как обработка тайм-аутов или отмена, отвечая на другие входные события или генерируя вывод. Мы рассмотрим соответствующий пример в следующем разделе.

Упражнение 8.8. Используя инструкцию `select`, добавьте к эхо-серверу из раздела 8.3 тайм-аут, чтобы он отключал любого клиента, который ничего не передает в течение 10 секунд.

8.8. Пример: параллельный обход каталога

В этом разделе мы создадим программу, которая сообщает об использовании дискового пространства одним или несколькими каталогами, указанными в командной строке наподобие команды Unix du. Бо́льшую часть работы выполняет показанная ниже функция walkDir, которая перечисляет все записи каталога dir с помощью вспомогательной функции dirents:

gopl.io/ch8/du1

```
// walkDir рекурсивно обходит дерево файлов с корнем в dir
// и отправляет размер каждого найденного файла в fileSizes.
func walkDir(dir string, fileSizes chan<- int64)
{
    for _, entry := range dirents(dir) {
        if entry.IsDir() {
            subdir := filepath.Join(dir, entry.Name())
            walkDir(subdir, fileSizes)
        } else {
            fileSizes <- entry.Size()
        }
    }
}

// dirents возвращает записи каталога dir.
func dirents(dir string) []os.FileInfo {
    entries, err := ioutil.ReadDir(dir)
    if err != nil {
        fmt.Fprintf(os.Stderr, "du1: %v\n", err)
        return nil
    }
    return entries
}
```

Функция ioutil.ReadDir возвращает срез значений os.FileInfo — ту же информацию, которую функция os.Stat возвращает для одного файла. Для каждого подкаталога walkDir рекурсивно вызывает сама себя, а для каждого файла walkDir отправляет сообщение в канал fileSizes. Сообщение представляет собой размер файла в байтах.

Показанная ниже функция main использует две go-подпрограммы. Фоновая go-подпрограмма вызывает walkDir для каждого каталога, указанного в командной строке, и в конце закрывает канал fileSizes. Главная go-подпрограмма вычисляет сумму размеров файлов, которые получает из канала, и в конце работы выводит итоговый результат.

```
// Команда du1 вычисляет суммарный размер всех файлов в каталоге.
package main

import (
```

```
    "flag"
    "fmt"
    "io/ioutil"
    "os"
    "path/filepath"
)

func main() {
    // Определяет исходные каталоги.
    flag.Parse()
    roots := flag.Args()
    if len(roots) == 0 {
        roots = []string{"."}
    }

    // Обход дерева файлов.
    fileSizes := make(chan int64)
    go func() {
        for _, root := range roots {
            walkDir(root, fileSizes)
        }
        close(fileSizes)
    }()

    // Вывод результатов.
    var nfiles, nbytes int64
    for size := range fileSizes {
        nfiles++
        nbytes += size
    }
    printDiskUsage(nfiles, nbytes)
}

func printDiskUsage(nfiles, nbytes int64) {
    fmt.Printf("%d файлов %.1f GB\n", nfiles, float64(nbytes)/1e9)
}
```

Эта программа надолго замирает перед тем, как вывести результат:

```
$ go build gopl.io/ch8/du1
$ ./du1 $HOME /usr /bin /etc
213201 файлов 62.7 GB
```

Программа будет выглядеть лучше, если будет информировать нас о ходе своей работы. Однако простое перемещение вызова `printDiskUsage` в цикл приведет к выводу тысяч строк.

Вариант `du`, приведенный ниже, периодически выводит итоговую величину, но только если указан флаг `-v`, поскольку не все пользователи хотят видеть сообщения о ходе выполнения. Фоновая go-подпрограмма, которая циклически обходит `roots`,

остается неизменной. Главная же go-подпрограмма теперь использует таймер для генерации событий каждые 500 мс и инструкцию `select`, чтобы ожидать либо сообщения с размером файла (в этом случае обновляются итоговые значения), либо события таймера (в этом случае выводятся текущие итоги). Если не указан флаг `-v`, канал `tick` остается нулевым, и его вариант в инструкции `select` становится отключенным.

gopl.io/ch8/du2

```
var verbose = flag.Bool("v",false,"вывод промежуточных результатов")
func main() {
    // ... запуск фоновой go-подпрограммы ...

    // Периодический вывод результатов.
    var tick <-chan time.Time
    if *verbose {
        tick = time.Tick(500 * time.Millisecond)
    }
    var nfiles, nbytes int64
loop:
    for {
        select {
            case size, ok := <-fileSizes:
                if !ok {
                    break loop // fileSizes был закрыт
                }
                nfiles++
                nbytes += size
            case <-tick:
                printDiskUsage(nfiles, nbytes)
        }
    }
    printDiskUsage(nfiles, nbytes) // Окончательные итоги
}
```

Поскольку программа больше не использует цикл по диапазону, первый вариант `select` должен явно проверять, не закрыт ли канал `fileSizes`, используя операцию получения с двумя результатами. Если канал был закрыт, выполнение цикла прерывается. Помеченная инструкция `break` обеспечивает выход как из инструкции `select`, так и из цикла; инструкция `break` без метки обеспечивала бы выход только из инструкции `select` и начало следующей итерации цикла.

Теперь программа дает нам неторопливый поток обновлений итоговых результатов:

```
$ go build gopl.io/ch8/du2
$ ./du2 v
$HOME /usr /bin /etc
28608 файлов 8.3 GB
54147 файлов 10.3 GB
93591 файлов 15.1 GB
127169 файлов 52.9 GB
```

```
175931 файлов 62.2 GB
213201 файлов 62.7 GB
```

Однако программа все еще работает слишком долго. Нет причин, по которым все вызовы `walkDir` нельзя выполнять параллельно, тем самым используя параллелизм для дисковой системы. Показанная ниже третья версия `du` создает новую go-подпрограмму для каждого вызова `walkDir`. Она использует `sync.WaitGroup` (раздел 8.5) для подсчета количества активных вызовов `walkDir` и go-подпрограмму, закрывающую канал `fileSizes`, когда счетчик становится равным нулю.

gopl.io/ch8/du3

```
func main() {
    // ...определение корней...

    // Параллельный обход каждого корня дерева файлов.
    fileSizes := make(chan int64)
    var n sync.WaitGroup
    for _, root := range roots {
        n.Add(1)
        go walkDir(root, &n, fileSizes)
    }
    go func() {
        n.Wait()
        close(fileSizes)
    }()
    // ... цикл select ...
}

func walkDir(dir string, n *sync.WaitGroup, fileSizes chan<- int64)
{
    defer n.Done()
    for _, entry := range dirents(dir) {
        if entry.IsDir() {
            n.Add(1)
            subdir := filepath.Join(dir, entry.Name())
            go walkDir(subdir, n, fileSizes)
        } else {
            fileSizes <- entry.Size()
        }
    }
}
```

Поскольку в пике программа создает многие тысячи go-подпрограмм, мы должны изменить функцию `dirents` так, чтобы она использовала подсчитывающий семафор для предотвращения открытия слишком большого количества файлов одновременно, так же, как мы делали для веб-сканера в разделе 8.6:

```
// sema — подсчитывающий семафор для ограничения параллельности.
var sema = make(chan struct{}, 20)
```

```
// dirents возвращает записи в каталоге dir.
func dirents(dir string) []os.FileInfo {
    sema <- struct{}{}        // Захват маркера
    defer func() { <-sema }() // Освобождение маркера
    // ...
```

Эта версия работает в несколько раз быстрее предыдущей, хотя ее ускорение сильно варьируется от системы к системе.

Упражнение 8.9. Напишите версию du, которая вычисляет и периодически выводит отдельные итоговые величины для каждого из каталогов `root`.

8.9. Отмена

Иногда нужно указать go-подпрограмме, что она должна прекратить свои действия, например, на веб-сервере, где она выполняет вычисления для клиента, соединение с которым разорвано.

Нет никакого способа, которым одна go-подпрограмма могла бы прекратить выполнение другой непосредственно, поскольку это допускало бы ситуацию, когда совместно используемые переменные могли оказаться в неопределенном состоянии. В программе запуска ракеты (раздел 8.7) мы отправляли одно значение в канал с именем `abort`, которое go-подпрограмма отсчета интерпретировала как запрос на прекращение работы. Но что если нам нужно отменить две go-подпрограммы или произвольное число go-подпрограмм?

Одна из возможностей может заключаться в отправлении стольких событий в канал прерывания, сколько go-подпрограмм следует отменить. Однако если некоторые из go-подпрограмм уже прекратили работу, наш счетчик окажется слишком большим, и наш отправитель остановится. С другой стороны, если эти go-подпрограммы породили другие go-подпрограммы, счетчик окажется слишком мал, и некоторые go-подпрограммы так и останутся в неведении относительно требующегося прекращения работы. В общем случае трудно знать, сколько go-подпрограмм работают от нашего имени в произвольный момент времени. Кроме того, go-подпрограмма, получив значение из канала `abort`, забирает это значение, и другие go-подпрограммы его не видят. Для отмены нам нужен надежный механизм *широковещания* события по каналу, чтобы многие go-подпрограммы могли его увидеть.

Вспомним, что после того, как канал был закрыт и опустошен от всех отправленных значений, последующие операции получения значения немедленно выдают нулевые значения. Мы можем использовать это свойство для создания механизма широковещания: не отправляйте значение в канал, а просто *закройте* его.

Мы можем добавить такую возможность в программу du из предыдущего раздела с помощью нескольких простых изменений. Сначала мы создаем канал отмены, в который не передаются никакие значения, но закрытие которого означает, что программа должна прекратить свою работу. Мы также определим вспомогательную функцию `cancelled`, которая проверяет, или *опрашивает*, состояние отмены при вызове.

```
gopl.io/ch8/du4
var done = make(chan struct{})

func cancelled() bool {
    select {
    case <- done:
        return true
    default:
        return false
    }
}
```

Далее мы создаем go-подпрограмму, которая будет считывать данные из стандартного ввода (обычно подключенного к терминалу). Как только будет считан любой ввод (например, пользователь нажмет клавишу <Enter>), данная go-подпрограмма сообщает об этом всем прочим go-подпрограммам путем закрытия канала done.

```
// Отмена обхода при обнаруженном вводе.
go func() {
    os.Stdin.Read(make([]byte, 1)) // Чтение одного байта
    close(done)
}()
```

Теперь нужно сделать наши go-подпрограммы реагирующими на сигнал отмены. В главной go-подпрограмме добавим в инструкцию `select` третий вариант, который пытается выполнить получение от канала done. При выборе этого варианта выполняется возврат из функции, но перед этим следует опустошить канал `fileSizes`, проигнорировав все значения до закрытия канала. Это делается для того, чтобы убедиться, что все активные вызовы `walkDir` могут выполняться до завершения без "зависания" при попытке отправления в канал `fileSizes`.

```
for {
    select {
    case <-done:
        // Опустошение канала fileSizes, чтобы позволить
        // завершиться существующим go-подпрограммам.
        for range fileSizes {
            // Ничего не делаем.
        }
        return
    case size, ok := <-fileSizes:
        // ...
    }
}
```

Go-подпрограмма `WalkDir` опрашивает состояние отмены, когда начинает работу, и, если это состояние установлено, возвращается, не выполняя никаких действий. Так все go-подпрограммы, созданные после отмены, превращаются в "пустышки":

```
func walkDir(dir string, n *sync.WaitGroup, fileSizes chan<-int64)
{
    defer n.Done()
    if cancelled() {
        return
    }
    for _, entry := range dirents(dir) {
        // ...
    }
}
```

Может оказаться выгодным опрос состояния отмены в цикле `walkDir`, чтобы избежать создания go-подпрограмм после события отмены. Отмена включает компромисс; быстрый ответ часто требует больших изменений в логике программы. Гарантия отсутствия дорогостоящих операций после отмены может потребовать обновления многих мест кода, но зачастую наибольшие выгоды могут быть получены от проверки отмены в нескольких важных местах.

Небольшое профилирование этой программы показало, что узким местом является захват маркера семафора в `dirents`. Инструкция `select`, показанная ниже, позволяет отменить эту операцию и уменьшает типичную задержку отмены программы от сотен миллисекунд до десятков:

```
func dirents(dir string) []os.FileInfo {
    select {
    case sema <-struct{}{}:   // Захват маркера
    case <-done:
        return nil            // Отмена
    }
    defer func() { <-sema }() // Освобождение маркера

    // ... Чтение каталога ...
}
```

Теперь, когда происходит отмена, все фоновые go-подпрограммы быстро останавливаются и происходит возврат из функции `main`. Конечно, когда происходит возврат из функции `main`, программа завершает работу, поэтому может быть трудно сказать, точно ли функция `main` очищает все после себя. Есть удобный трюк, который можно использовать в ходе тестирования: если вместо возврата из функции `main` в случае отмены мы выполним вызов `panic`, то среда выполнения создаст дамп стека каждой go-подпрограммы в программе. Если остается только одна главная go-подпрограмма, то она выполняет всю очистку. Но если остаются другие go-подпрограммы, они могут не быть корректно отменены или, возможно, их отмена требует времени. В этом случае может быть целесообразным небольшое расследование. Дамп аварийной ситуации часто содержит достаточную информацию, чтобы различить эти случаи.

Упражнение 8.10. Запросы HTTP могут быть отменены с помощью закрытия необязательного канала `Cancel` в структуре `http.Request`. Измените веб-сканер из раздела 8.6 так, чтобы он поддерживал отмену.

Указание. Функция `http.Get` не позволяет настроить `Request`. Вместо этого создайте запрос с использованием `http.NewRequest`, установите его поле `Cancel` и выполните запрос с помощью вызова `http.DefaultClient.Do(req)`.

Упражнение 8.11. Следуя подходу `mirroredQuery` из раздела 8.4.4, реализуйте вариант программы `fetch`, который параллельно запрашивает несколько URL. Как только получен первый ответ, прочие запросы отменяются.

8.10. Пример: чат-сервер

Мы завершим эту главу разработкой чат-сервера, который позволяет нескольким пользователям обмениваться текстовыми сообщениями друг с другом. В этой программе есть четыре вида go-подпрограмм. Имеется по экземпляру go-подпрограмм `main` и `broadcaster`, а для каждого подключенного клиента имеется по одной go-подпрограмме `handleConn` и `clientWriter`. Go-подпрограмма `broadcaster` является хорошей иллюстрацией использования инструкции `select`, так как она должна реагировать на три различных вида сообщений.

Работа главный go-подпрограммы, показанной ниже, состоит в прослушивании и приеме входящих сетевых подключений от клиентов. Для каждого из них она создает новую go-подпрограмму `handleConn`, так же как это делал параллельный эхо-сервер в начале этой главы.

gopl.io/ch8/chat

```
func main() {
    listener, err := net.Listen("tcp", "localhost:8000")
    if err != nil {
        log.Fatal(err)
    }

    go broadcaster()
    for {
        conn, err := listener.Accept()
        if err != nil {
            log.Print(err)
            continue
        }
        go handleConn(conn)
    }
}
```

Теперь перейдем к go-подпрограмме `broadcaster`. Ее локальная переменная `clients` записывает текущее множество подключенных клиентов. Единственная информация, записываемая о каждом клиенте, — это его канал для исходящих сообщений, о котором будет сказано чуть позже.

```
type client chan<- string          // Канал исходящих сообщений
var (
    entering = make(chan client)
```

```
    leaving  = make(chan client)
    messages = make(chan string) // Все входящие сообщения клиента
)

func broadcaster() {
    clients := make(map[client]bool) // Все подключенные клиенты
    for {
        select {
        case msg := <-messages:
            // Широковещательное входящее сообщение во все
            // каналы исходящих сообщений для клиентов.
        for cli := range clients {
            cli <-msg
        }
        case cli := <-entering:
            clients[cli] = true
        case cli := <-leaving:
            delete(clients, cli)
            close(cli)
        }
    }
}
```

Go-подпрограмма широковещателя прослушивает глобальные каналы `entering` и `leaving` в поисках объявлений о поступающих и убывающих клиентах. Получив информацию об одном из этих событий, она обновляет множество `clients` и, если событие — убытие клиента, закрывает канал исходящих сообщений клиента. Широковещатель также следит за событиями в глобальном канале `messages`, в который каждый клиент отправляет все входящие сообщения. Широковещатель, получив одно из этих событий, передает сообщение каждому подключенному клиенту.

Теперь давайте рассмотрим go-подпрограммы каждого клиента. Функция `handleConn` создает новый канал исходящих сообщений для своего клиента и объявляет широковещателю о поступлении этого клиента по каналу `entering`. Затем она считывает каждую строку текста от клиента, отправляя каждую строку широковещателю по глобальному каналу входящих сообщений и предваряя каждое сообщение указанием отправителя. Когда от клиента получена вся информация, `handleConn` объявляет об убытии клиента по каналу `leaving` и закрывает подключение.

```
func handleConn(conn net.Conn) {
    ch := make(chan string)    // Исходящие сообщения клиентов
    go clientWriter(conn, ch)

    who := conn.RemoteAddr().String()
    ch <- "Вы " + who
    messages <- who + " подключился"
    entering <- ch

    input := bufio.NewScanner(conn)
```

```
    for input.Scan() {
        messages <- who + ": " + input.Text()
    }
    // Примечание: игнорируем потенциальные ошибки input.Err()

    leaving <- ch
    messages <- who + " отключился"
    conn.Close()
}

func clientWriter(conn net.Conn, ch <-chan string) {
    for msg := range ch {
        fmt.Fprintln(conn, msg) // Примечание: игнорируем ошибки сети
    }
}
```

Кроме того, `handleConn` создает go-подпрограмму `clientWriter` для каждого клиента, которая получает широковещательные сообщения по исходящему каналу клиента и записывает их в сетевое подключение клиента. Цикл завершается, когда широковещатель закрывает канал после получения уведомления `leaving`.

Приведенный ниже вывод показывает сервер в действии с двумя клиентами в отдельных окнах на одном и том же компьютере с использованием `netcat` для чата:

```
$ go build gopl.io/ch8/chat
$ go build gopl.io/ch8/netcat3
$ ./chat &
$ ./netcat3
Вы 127.0.0.1:64208                  $ ./netcat3
127.0.0.1:64211 подключился         Вы 127.0.0.1:64211
Привет!
127.0.0.1:64208: Привет!            127.0.0.1:64208: Привет!
                                    И вам привет.
127.0.0.1:64211: И вам привет.      127.0.0.1:64211: И вам привет.
^C
                                    127.0.0.1:64208 отключился
$ ./netcat3
Вы 127.0.0.1:64216                  127.0.0.1:64216 подключился
                                    Здравствуйте.
127.0.0.1:64211: Здравствуйте.      127.0.0.1:64211: Здравствуйте.
                                    ^C
127.0.0.1:64211 отключился
```

Во время сеанса чата для n клиентов эта программа запускает $2n + 2$ параллельных сообщающихся между собой go-подпрограмм, но она не нуждается в явных операциях блокировки (раздел 9.2). Отображение `clients` ограничено одной широковещательной go-подпрограммой, поэтому к нему не может выполняться параллельный доступ. Единственными переменными, которые разделяются несколькими go-подпрограммами, являются каналы и экземпляры `net.Conn`, но и те, и другие яв-

ляются *безопасными с точки зрения параллелизма*. Более подробно о вопросах безопасности, ограничения и следствий из совместного использования переменных go-подпрограммами мы поговорим в следующей главе.

Упражнение 8.12. Заставьте широковещательную go-подпрограмму сообщать текущее множество клиентов каждому вновь подключенному клиенту. Для этого требуется, чтобы множество `clients` и каналы `entering` и `leaving` записывали также имена клиентов.

Упражнение 8.13. Заставьте сервер отключать простаивающих клиентов, которые не прислали ни одного сообщения за последние 5 минут.

Указание: вызов `conn.Close()` в другой go-подпрограмме деблокирует активный вызов `Read`, такой, как выполняемый вызовом `input.Scan()`.

Упражнение 8.14. Измените сетевой протокол чат-сервера так, чтобы каждый клиент предоставлял при подключении свое имя. Используйте это имя вместо сетевого адреса в префиксе сообщения.

Упражнение 8.15. Ошибка своевременного чтения данных любой клиентской программой в конечном итоге вызывает сбой всех клиентских программ. Измените широковещатель таким образом, чтобы в случае, когда go-подпрограмма передачи сообщения клиентам не готова принять сообщение, он вместо ожидания пропускал сообщение. Кроме того, добавьте буферизацию каждого канала исходящих сообщений клиента, чтобы большинство сообщений не удалялись. Широковещатель должен использовать неблокирующее отправление в этот канал.

9

Параллельность и совместно используемые переменные

В предыдущей главе мы представили несколько программ, которые используют go-подпрограммы и каналы для выражения параллелизма непосредственным и естественным путем. Однако, поступая так, мы затушевывали целый ряд важных и тонких вопросов, которые программисты должны учитывать при написании параллельного кода.

В этой главе мы более детально рассмотрим механику параллелизма. В частности, мы укажем на некоторые из проблем, связанных с совместным использованием переменных несколькими go-подпрограммами, и рассмотрим аналитические методы распознавания этих проблем и шаблоны их решений. Наконец мы поясним некоторые технические различия между go-подпрограммами и потоками операционной системы.

9.1. Состояния гонки

В последовательной программе, т.е. в программе с единственной go-подпрограммой, все этапы программы выполняются в знакомом порядке, определяемом логикой программы. Например, в последовательности инструкций первая выполняется до второй и т.д. В программе с двумя или более go-подпрограммами этапы каждой go-подпрограммы выполняются в знакомом порядке, но в общем случае мы не знаем, предшествует ли событие x в одной go-подпрограмме событию y в другой go-подпрограмме, происходит ли оно после x или одновременно с ним. Если мы не можем уверенно сказать, что одно событие *предшествует* другому, такие события x и y являются *параллельными*.

Рассмотрим функцию, которая правильно работает в последовательной программе. Она является *безопасной с точки зрения параллельности*, если продолжает работать правильно даже при параллельном вызове, т.е. при вызове из двух или более go-подпрограмм без какой бы то ни было дополнительной синхронизации. Мы можем обобщить это понятие для множества сотрудничающих функций, таких как методы и операции определенного типа. Тип является безопасным с точки зрения параллельности, если все доступные методы и операции являются таковыми.

Можно сделать программу безопасной с точки зрения параллельности, не делая каждый конкретный тип в этой программе таковым. В действительности безопасные с точки зрения параллельности типы являются скорее исключением, чем правилом, так что получать параллельный доступ к переменной следует, только если в документации для ее типа сказано, что это безопасно. Мы избегаем одновременного доступа к большинству переменных, *ограничивая* их одной go-подпрограммой или поддерживая высокоуровневый инвариант *взаимного исключения*. Мы поясним эти термины далее в этой главе.

В противоположность этому от экспортируемых функций уровня пакета обычно ожидается безопасность с точки зрения параллельности. Поскольку переменные уровня пакета не могут быть ограничены одной go-подпрограммой, функции, которые их изменяют, должны обеспечивать взаимное исключение.

Существует множество причин, по которым функция может не работать при параллельном вызове, включая взаимоблокировки, динамические взаимоблокировки и голодание. У нас не хватит места для обсуждения всех этих явлений, поэтому мы сосредоточимся на наиболее важном — *состоянии гонки*.

Состояние гонки — это ситуация, в которой программа не дает правильный результат для некоторого чередования операций нескольких go-подпрограмм. Состояния гонки крайне вредны, поскольку могут оставаться скрытыми в программе и проявляться очень редко, возможно, только при большой нагрузке или при использовании некоторых компиляторов, на определенных платформах или для конкретной архитектуры. Это делает их сложными для воспроизведения и диагностики.

Традиционно серьезность состояния гонки поясняется через метафору финансовых потерь, поэтому мы рассмотрим простую программу для работы с банковскими счетами:

```
// Пакет bank реализует банк с единственным счетом.
package bank

var balance int

func Deposit(amount int) { balance = balance + amount }

func Balance() int { return balance }
```

(Мы могли бы написать тело функции `Deposit` как `balance += amount`, что было бы эквивалентно, но более длинная запись позволит упростить объяснение.)

Для такой тривиальной программы очевидно с первого взгляда, что любая последовательность вызовов `Deposit` и `Balance` даст правильный ответ, т.е. `Balance` будет сообщать о верной сумме всех положенных в банк вкладов. Однако, если мы вызываем эти функции не последовательно, а одновременно, больше не гарантируется, что `Balance` даст правильный ответ. Рассмотрим две следующие go-подпрограммы, которые представляют собой две операции с общим банковским счетом:

```
// Алиса:
go func() {
```

```
    bank.Deposit(200)                // A1
    fmt.Println("=", bank.Balance()) // A2
}()

// Боб:
go bank.Deposit(100)                 // B
```

Алиса вкладывает 200 долларов, а затем проверяет свой баланс, в то время как Боб вкладывает 100 долларов. Поскольку шаги `A1` и `A2` выполняются одновременно с `B`, мы не можем предсказать точный их порядок. Интуитивно может показаться, что есть только три возможности упорядочения, которые мы будем называть "Алиса первая", "Боб первый" и "Алиса/Боб/Алиса". В следующей таблице показаны значения переменной `balance` после каждого шага (строки в кавычках представляют вывод баланса на печать):

```
Алиса первая     Боб первый     Алиса/Боб/Алиса
           0              0                   0
A1       200    B       100        A1       200
A2   "= 200"    A1      300         B       300
 B       300    A2   "= 300"        A2   "= 300"
```

Во всех случаях окончательный баланс составляет 300 долларов. Единственной вариацией является то, включает ли баланс Алисы транзакции Боба, но в любом случае клиенты удовлетворены.

Но интуиция нас обманывает. Есть и четвертый возможный результат, в котором вклад Боба происходит посреди вклада Алисы, после прочтения значения баланса (`balance + amount`), но до его обновления (`balance = ...`), приводя к исчезновению транзакции Боба. Дело в том, что операция вклада Алисы `A1` на самом деле представляет собой последовательность двух операций, чтения и записи; назовем их `A1r` и `A1w`. Вот как выглядит чередование, вызывающее проблемы:

```
Состояние гонки
           0
A1r        0          ... = balance + amount
B        100
A1w      200          balance = ...
A2    "= 200"
```

После `A1r` выражение `balance + amount` равно 200, и именно это значение записывается во время `A1w`, несмотря на выполненный вклад. Окончательный баланс составляет всего 200 долларов. Банк стал на 100 долларов богаче за счет Боба.

Эта программа содержит разновидность состояния гонки, которая называется *гонкой данных*. Гонка данных осуществляется, когда две go-подпрограммы одновременно обращаются к одной и той же переменной и по крайней мере одно из обращений является записью.

Все оказывается еще хуже, если гонка данных включает переменную типа, большего, чем одно машинное слово, такого как интерфейс, строка или срез. Приведен-

ный далее код обновляет значение переменной x одновременно двумя срезами разной длины:

```
var x []int
go func() { x = make([]int, 10) }()
go func() { x = make([]int, 1000000) }()
x[999999] = 1 // Примечание: неопределенное поведение;
              // возможно повреждение памяти!
```

Значение x в последней инструкции не определено; это может быть nil, срез длиной 10 или срез длиной 1 000 000. Но вспомните, что у среза есть три составные части: указатель, длина и емкость. Если указатель получен из первого вызова make, а длина — из второго, x становится химерой[1], срезом, длина которого составляет 1 000 000, но базовый массив имеет только 10 элементов. В этом случае сохранение 999 999-го элемента выполняет запись в некоторой далеко отстоящей ячейке памяти с последствиями, которые невозможно предсказать и трудно локализовать и отлаживать. Это семантическое минное поле называется *неопределенным поведением* и хорошо известно программистам на языке C; к счастью, в Go такие неприятности встречаются куда реже, чем в C.

Даже интуитивная идея, что параллельная программа представляет собой чередование нескольких последовательных программ, является ложной. Как мы увидим в разделе 9.4, гонка данных может приводить к еще более странным результатам. Многие программисты — даже некоторые из самых умных — иногда предлагают обоснования для допуска известной гонки данных в своей программе: "стоимость взаимного исключения слишком высока", "эта логика — только для протоколирования", "я не возражаю против потери некоторых сообщений" и т.д. Отсутствие проблем при использовании данных компилятора и платформы может дать ложную уверенность в благополучности ситуации. Хорошее практическое правило гласит, что *не существует такого понятия, как доброкачественная гонка данных*. Так как же избегать гонки данных в наших программах?

Мы повторим определение, поскольку это очень важно: гонка данных осуществляется, когда две go-подпрограммы одновременно обращаются к одной и той же переменной и по крайней мере одно из обращений представляет собой запись. Из этого определения следует, что существует три способа избежать гонки данных.

Первый способ — не записывать переменную. Рассмотрим приведенное ниже отображение с отложенным заполнением при первом запросе ключа. Если Icon вызывается последовательно, программа работает нормально, но если Icon вызывается параллельно, при обращении к отображению имеется гонка данных.

```
var icons = make(map[string]image.Image)

func loadIcon(name string) image.Image

// Применание: небезопасно с точки зрения параллельности!
```

[1] В данном случае имеется в виду химера как мифологическое существо, созданное в результате объединения частей различных животных. — *Примеч. пер.*

```
func Icon(name string) image.Image {
    icon, ok := icons[name]
    if !ok {
        icon = loadIcon(name)
        icons[name] = icon
    }
    return icon
}
```

Если вместо этого мы инициализируем отображение всеми необходимыми записями перед созданием дополнительных go-подпрограмм и никогда не будем изменять его снова, то любое количество go-подпрограмм может безопасно одновременно вызывать `Icon`, поскольку каждая из них только читает отображение.

```
var icons = map[string]image.Image{
    "spades.png"  : loadIcon("spades.png"),
    "hearts.png"  : loadIcon("hearts.png"),
    "diamonds.png": loadIcon("diamonds.png"),
    "clubs.png"   : loadIcon("clubs.png"),
}
// Безопасно с точки зрения параллельности.
func Icon(name string) image.Image { return icons[name] }
```

В приведенном выше примере переменная `icons` присваивается во время инициализации пакета, которая *происходит до* того, как запускается функция `main` программы. После инициализации `icons` никогда не изменяется. Структуры данных, которые никогда не изменяются или являются неизменяемыми по своей сути, безопасны с точки зрения параллельности и не требуют синхронизации. Но очевидно, что мы не можем использовать этот подход, если обновления играют существенную роль, как в случае банковского счета.

Второй способ избежать гонки данных — избегать обращения к переменной из нескольких go-подпрограмм. Это подход, использованный многими из программ предыдущей главы. Например, главная go-подпрограмма в параллельном веб-сканере (раздел 8.6) является единственной go-подпрограммой, обращающейся к отображению `seen`, а go-подпрограмма `broadcaster` чат-сервера (раздел 8.10) является единственной go-подпрограммой, обращающейся к отображению `clients`. Эти переменные *ограничены* одной go-подпрограммой.

Поскольку другие go-подпрограммы не могут получить непосредственный доступ к переменной, они должны использовать канал для запроса у ограничивающей go-подпрограммы получения значения или обновления переменной. Это то, что подразумевается мантрой Go "не связывайтесь путем совместного использования памяти; совместно используйте память путем связи". Go-подпрограмма, посредничающая в доступе к ограниченной переменной с использованием запроса по каналу, называется *go-подпрограммой управления (монитором)* для данной переменной. Например, go-подпрограмма `broadcaster` управляет доступом к отображению `clients`.

Вот пример банка, переписанный с переменной `balance`, которая ограничивается монитором `teller`:

```
gopl.io/ch9/bank1
// Пакет bank предоставляет безопасный с точки зрения
// параллельности банк с одним счетом.
package bank

var deposits = make(chan int) // Отправление вклада
var balances = make(chan int) // Получение баланса

func Deposit(amount int) { deposits <- amount }
func Balance() int       { return <- balances }

func teller() {
    var balance int // balance ограничен go-подпрограммой teller
    for {
        select {
        case amount := <- deposits:
            balance += amount
        case balances <- balance:
        }
    }
}

func init() {
    go teller()     // Запуск управляющей go-подпрограммы
}
```

Даже когда переменная не может быть ограничена одной go-подпрограммой на все ее время жизни, ограничение все же может быть решением проблемы параллельного доступа. Например, распространено совместное использование переменной go-подпрограммами в конвейере путем передачи ее адреса от одной стадии к другой по каналу. Если каждый этап конвейера воздерживается от доступа к переменной после ее отправки на следующий этап, все обращения к такой переменной последовательны. По сути, переменная ограничена одной стадией конвейера, а затем ограничивается следующей и т.д. Такой подход иногда называют последовательным ограничением.

В приведенном ниже примере `Cake` последовательно ограничены, сначала go-подпрограммой `baker`, затем — go-подпрограммой `icer`:

```
type Cake struct{ state string }

func baker(cooked chan<- *Cake) {
    for {
        cake := new(Cake)
        cake.state = "cooked"
        cooked <- cake  // baker больше не будет работать с этим cake
    }
}

func icer(iced chan<- *Cake, cooked <-chan *Cake) {
```

```
    for cake := range cooked {
        cake.state = "iced"
        iced <- cake    // icer больше не будет работать с этим cake
    }
}
```

Третий способ избежать гонки данных состоит в том, чтобы позволить многим go-подпрограммам обращаться к переменной, но только по одной за раз. Этот подход известен как *взаимное исключение* и является темой следующего раздела.

Упражнение 9.1. Добавьте функцию снятия со счета `Withdraw(amount int) bool` в программу `gopl.io/ch9/bank1`. Результат должен указывать, прошла ли транзакция успешно или произошла ошибка из-за нехватки средств. Сообщение, отправляемое go-подпрограмме монитора, должно содержать как снимаемую сумму, так и новый канал, по которому go-подпрограмма монитора сможет отправить булев результат функции `Withdraw`.

9.2. Взаимные исключения: sync.Mutex

В разделе 8.6 мы использовали буферизованный канал в качестве *подсчитывающего семафора*, чтобы гарантировать, что HTTP-запросы одновременно будут выполнять не более 20 go-подпрограмм. Мы используем ту же самую идею и канал емкостью 1 для того, чтобы гарантировать, что одновременно к совместно используемой переменной может обратиться только одна go-подпрограмма. Семафор, которым ведет подсчет только до 1, называется *бинарным семафором*.

gopl.io/ch9/bank2

```
var (
    sema = make(chan struct{},1) // Бинарный семафор для
    balance int                  // защиты balance
)

func Deposit(amount int) {
    sema <- struct{}{}           // Захват маркера
    balance = balance + amount
    <- sema                      // Освобождение маркера
}

func Balance() int {
    sema <- struct{}{}           // Захват маркера
    b := balance
    <- sema                      // Освобождение маркера
    return b
}
```

Такой шаблон *взаимного исключения* настолько полезен, что поддерживается непосредственно типом `Mutex` из пакета `sync`. Его метод `Lock` захватывает маркер (вызывает *блокировку*), а метод `Unlock` его освобождает:

```
gopl.io/ch9/bank3
import "sync"

var (
    mu sync.Mutex   // Защищает balance
    balance int
)

func Deposit(amount int) {
    mu.Lock()
    balance = balance + amount
    mu.Unlock()
}

func Balance() int {
    mu.Lock()
    b := balance
    mu.Unlock()
    return b
}
```

Каждый раз, когда go-подпрограмма обращается к переменным банка (здесь только `balance`), она должна вызвать метод `Lock` мьютекса, чтобы получить монопольную блокировку. Если блокировка выполнена некоторой другой go-подпрограммой, текущая операция будет заблокирована до тех пор, пока другая go-подпрограмма не вызовет `Unlock` и не разрешит тем самым блокировку другим go-подпрограммам. Мьютекс охраняет совместно используемые переменные. По соглашению охраняемые мьютексом переменные объявляются сразу же после объявления самого мьютекса. Если вы поступаете иначе, обязательно документируйте это отступление от правил.

Область кода между `Lock` и `Unlock`, в которой go-подпрограмма может свободно читать и модифицировать совместно используемые переменные, называется *критическим разделом*. Вызов `Unlock` владельцем блокировки *предшествует* блокировке любой другой go-подпрограммой. Важно, чтобы go-подпрограмма обязательно освобождала блокировку на всех путях выполнения функции, включая пути ошибок.

Приведенная выше программа банка иллюстрирует распространенный шаблон параллелизма. Набор экспортируемых функций инкапсулирует одну или несколько переменных, так что единственный способ доступа к переменным — через эти функции (или методы для переменных объекта). Каждая функция выполняет блокировку в начале и освобождение от нее в конце, обеспечивая тем самым недоступность совместно используемых переменных одновременно нескольким go-подпрограммам. Этот механизм работы с блокировками, функциями и переменными называется *монитором*.

Поскольку критические разделы функций `Deposit` и `Balance` очень коротки — в одну строку, без ветвлений, — вызов `Unlock` в конце прост. В более сложных критических разделах, в особенности в тех, в которых ошибки приводят к более раннему

возвращению из программы, может быть трудно гарантировать, что вызовы Lock и Unlock строго соответствуют один другому на всех путях выполнения. На помощь приходит инструкция defer: откладывание вызова Unlock неявно расширяет критический раздел до конца текущей функции, что избавляет нас от необходимости помнить о вставке вызова Unlock в одном или нескольких местах вдали от вызова блокировки.

```
func Balance() int {
    mu.Lock()
    defer mu.Unlock()
    return balance
}
```

В приведенном выше примере Unlock выполняется *после* того, как инструкция return считывает значение переменной balance, так что функция Balance безопасна с точки зрения параллельности. В качестве бонуса мы больше не нуждаемся в локальной переменной b.

Кроме того, отсроченный вызов Unlock будет выполняться даже при аварийной ситуации в критическом разделе, что может иметь важное значение в программах, которые используют recover (раздел 5.10). Применение defer немного дороже, чем явный вызов Unlock, но недостаточно, чтобы оправдать менее ясный код. В параллельных программах всегда следует предпочитать ясность и сопротивляться преждевременной оптимизации. Там, где это возможно, используйте defer и позвольте критическому разделу продлиться до конца функции.

Рассмотрим приведенную ниже функцию Withdraw. При успешном завершении она уменьшает баланс на указанную величину и возвращает true. Но если на счету недостаточно средств для транзакции, Withdraw восстанавливает старое значение баланса и возвращает значение false.

```
// Примечание: функция не атомарная!
func Withdraw(amount int) bool {
    Deposit(amount)
    if Balance() < 0 {
        Deposit(amount)
        return false // Недостаточно средств
    }
    return true
}
```

Функция в конечном итоге дает правильный результат, но имеет неприятный побочный эффект. При попытке снятия со счета чрезмерной суммы баланс временно опускается ниже нуля. Это может привести к тому, что параллельное снятие скромной суммы окажется ложно отвергнутым. Так что, если Боб попытается купить спортивный автомобиль, Алиса не сможет оплатить свой утренний кофе. Проблема заключается в том, что функция Withdraw не атомарная: она состоит из последовательности трех отдельных операций, каждая из которых захватывает, а затем освобождает мьютекс, но ничто не блокирует всю последовательность.

В идеале функция `Withdraw` должна выполнить блокировку один раз для всех операций. Однако такая попытка не работает

```
// Примечание: неверно!
func Withdraw(amount int) bool {
    mu.Lock()
    defer mu.Unlock()
    Deposit(-amount)
    if Balance() < 0 {
        Deposit(amount)
        return false // Недостаточно средств
    }
    return true
}
```

Функция `Deposit` пытается захватить мьютекс второй раз путем вызова `mu.Lock()`, но так как блокировка мьютекса не *реентерабельна* (не позволяет блокировать уже заблокированный мьютекс), это приводит к взаимоблокировке, когда программа не выполняет никаких действий, поскольку функция `Withdraw` заблокирована навечно.

Не реентерабельность мьютексов вполне понятна и обоснована. Мьютексы призваны обеспечить сохранение определенных инвариантов совместно используемых переменных в критических точках во время выполнения программы. Один из этих инвариантов — "никакая go-подпрограмма не имеет доступа к совместно используемой переменной", но могут быть и дополнительные инварианты, специфичные для структуры данных, которую охраняет мьютекс. Когда go-подпрограмма выполняет блокировку, она может предполагать выполнение инвариантов. Во время блокировки она может обновлять совместно используемые переменные, так что инварианты временно нарушаются. Однако при снятии блокировки go-подпрограмма должна гарантировать, что порядок восстановлен и инварианты вновь выполняются. Хотя реентерабельный мьютекс мог бы гарантировать, что никакие другие go-подпрограммы не получают доступ к совместно используемым переменным, никакие другие дополнительные инварианты этих переменных он бы защитить не мог.

Распространенное решение заключается в разделении функций, таких как `Deposit`, на две: неэкспортируемую функцию `deposit`, которая предполагает, что блокировка уже выполнена и выполняет реальную работу, и экспортируемую функцию `Deposit`, которая выполняет блокировку до вызова `deposit`. Тогда мы можем выразить функцию `Withdraw` с помощью функции `deposit` следующим образом:

```
func Withdraw(amount int) bool {
    mu.Lock()
    defer mu.Unlock()
    deposit(-amount)
    if balance < 0 {
        deposit(amount)
        return false // Недостаточно средств
    }
```

```
    return true
}

func Deposit(amount int) {
    mu.Lock()
    defer mu.Unlock()
    deposit(amount)
}

func Balance() int {
    mu.Lock()
    defer mu.Unlock()
    return balance
}

// Эта функция требует, чтобы была выполнена блокировка.
func deposit(amount int) { balance += amount }
```

Конечно, приведенная здесь функция `deposit` настолько тривиальна, что функции `Withdraw` нет смысла ее вызывать; тем не менее она иллюстрирует описанный выше принцип.

Инкапсуляция (раздел 6.6), уменьшая неожиданные взаимодействия в программе, помогает нам поддерживать инварианты структур данных. По той же причине инкапсуляция помогает также поддерживать инварианты параллелизма. При использовании мьютекса убедитесь, что и он, и переменные, которые он защищает, не экспортируются, независимо от того, являются ли они переменными уровня пакета или полями структуры.

9.3. Мьютексы чтения/записи: sync.RWMutex

Увидев внезапное исчезновение 100 долларов, Боб пишет программу для проверки своего баланса в банке сотни раз в секунду. Он запускает ее у себя дома, на работе и на своем телефоне. Банк замечает, что увеличение трафика замедляет вклады и снятия, так как все запросы `Balance` выполняются последовательно, выполняя исключительную блокировку и временно препятствуя выполнению других go-подпрограмм.

Поскольку функции `Balance` нужно только *читать* состояние переменной, множественные параллельные вызовы этой функции в действительности будут безопасными, если только не выполняется вызов `Deposit` или `Withdraw`. При таком сценарии нам нужен особый вид блокировки, который позволяет операциям чтения выполняться параллельно друг с другом, но операции записи получают полностью исключительный доступ. Такая блокировка называется *несколько читателей, один писатель* и в Go обеспечивается мьютексом `sync.RWMutex`:

```
var mu sync.RWMutex
var balance int

func Balance() int {
```

```
    mu.RLock()    // Блокировка читателя
    defer mu.RUnlock()
    return balance
}
```

Теперь функция `Balance` вызывает методы `RLock` и `RUnlock` для захвата и освобождения блокировки *читателя*, или *неисключительной* блокировки. Функция `Deposit`, оставшаяся неизмененной, вызывает методы `mu.Lock` и `mu.Unlock` для захвата и освобождения блокировки *писателя*, или *исключительной* блокировки.

После этого изменения большинство запросов Боба будут выполняться параллельно один другому и заканчиваться более быстро. Блокировка будет доступна дольше, и запросы `Deposit` смогут выполняться своевременно.

`RLock` может использоваться, только если в критическом разделе нет записи совместно используемых переменных. В общем случае мы не должны полагаться на то, что *логически* предназначенные только для чтения функции или методы не обновляют также некоторые переменные. Например, метод, который представляется простым методом получения значения, может увеличить и значение счетчика внутреннего использования, или обновить кеш, чтобы повторные вызовы выполнялись быстрее. Если вы сомневаетесь, используйте исключительную блокировку `Lock`.

Использовать `RWMutex` выгодно только тогда, когда большинству go-подпрограмм требуется блокировка читателей, и за блокировку ведется состязание, т.е. go-подпрограммам приходится регулярно ожидать для ее захвата. `RWMutex` требует более сложной внутренней бухгалтерии, что делает его медленнее обычных мьютексов.

9.4. Синхронизация памяти

Вы можете удивиться, почему метод `Balance` требует взаимного исключения, основанного на канале или мьютексе. В конце концов, в отличие от `Deposit`, он состоит только из одной операции, поэтому нет опасности выполнения другой go-подпрограммы "посредине" него. Тому есть две причины. Первая заключается в том, что не менее важно, чтобы метод `Balance` также не выполнялся посреди некоторых других операций наподобие `Withdraw`. Вторая (и более тонкая) причина в том, что синхронизация представляет собой нечто большее, чем просто порядок выполнения нескольких go-подпрограмм; синхронизация также влияет на память.

В современном компьютере могут быть десятки процессоров, каждый из которых имеет собственный локальный кеш оперативной памяти. Для повышения эффективности запись в память буферизуется в пределах каждого процессора и сбрасывается в основную память только по необходимости. Записи в основную память могут быть выполнены не в таком порядке, в каком они выполнялись записывающими go-подпрограммами. Такие примитивы синхронизации, как коммуникационный канал и операции мьютекса, заставляют процессор выполнить сброс и зафиксировать все накопленные записи, так что результаты выполнения go-подпрограмм до этой точки гарантированно будут видимыми для go-подпрограмм, работающих на других процессорах.

Рассмотрим возможные выводы следующего фрагмента кода:

```
var x, y int
go func() {
    x = 1                   // A1
    fmt.Print("y:", y, " ") // A2
}()
go func() {
    y = 1                   // B1
    fmt.Print("x:", x, " ") // B2
}()
```

Поскольку эти две go-подпрограммы параллельны и обращаются к совместно используемым переменным без взаимного исключения, возникает гонка данных. Поэтому мы не должны удивляться тому, что программа не является детерминированной. Мы могли бы ожидать любого из четырех приведенных ниже результатов, которые соответствуют интуитивному представлению о чередовании инструкций программы:

```
y:0 x:1
x:0 y:1
x:1 y:1
y:1 x:1
```

Четвертая строка, например, может быть пояснена последовательностью `A1,B1,A2,B2` или последовательностью `B1,A1,A2,B2`. Однако следующие два вывода могут показаться удивительными:

```
x:0 y:0
y:0 x:0
```

Однако в зависимости от компилятора, процессора и многих других факторов они также могут иметь место. Какой же последовательностью чередования четырех инструкций можно пояснить такие результаты?

В пределах одной go-подпрограммы результаты работы каждой инструкции гарантированно осуществляются в порядке исполнения; go-подпрограммы *последовательно согласованны*. Однако в отсутствие явной синхронизации с использованием канала или мьютекса нет никакой гарантии, что события видимы всем go-подпрограммам в одном и том же порядке. Хотя go-подпрограмма *A* должна наблюдать результат записи `x = 1` до чтения значения `y`, она не обязательно будет наблюдать запись `y`, сделанную go-подпрограммой *B*, поэтому *A* может вывести *устаревшее* значение `y`.

Соблазнительно попытаться понять параллелизм так, как будто он соответствует *некоторому* чередованию инструкций каждой go-подпрограммы, но, как показано в примере выше, современные компиляторы или процессоры работают не совсем так. Поскольку присваивание и вызов `Print` относятся к разным переменным, компилятор может посчитать, что порядок этих двух инструкций не может повлиять на результат, и поменять их местами. Если две go-подпрограммы выполняются на разных процессорах, каждый со своей кеш-памятью, запись в память одной go-подпрограммой не видима для вызова `Print` другой go-подпрограммы до тех пор, пока кеши не будут синхронизированы с основной памятью.

Всех этих проблем параллелизма можно избежать путем последовательного, согласованного применения простых и давно известных шаблонов. Там, где это возможно, ограничьте переменные одной go-подпрограммой; для всех других переменных используйте взаимные исключения.

9.5. Отложенная инициализация: sync.Once

Отложить этап дорогостоящей инициализации до того момента, когда она необходима, — хорошая практика. Инициализация переменных в начале работы увеличивает задержку запуска программы и не является необходимой, если выполнение программы не всегда достигает той части, которая использует эту переменную. Давайте вернемся к переменной `icons`, которую мы уже видели в этой главе:

```
var icons map[string]image.Image
```

Следующая версия `Icon` использует *отложенную инициализацию*:

```
func loadIcons() {
    icons = map[string]image.Image{
        "spades.png":   loadIcon("spades.png"),
        "hearts.png":   loadIcon("hearts.png"),
        "diamonds.png": loadIcon("diamonds.png"),
        "clubs.png":    loadIcon("clubs.png"),
    }
}

// Примечание: не безопасно с точки зрения параллельности!
func Icon(name string) image.Image {
    if icons == nil {
        loadIcons()   // Однократная инициализация
    }
    return icons[name]
}
```

Для переменной, доступ к которой осуществляется только одной go-подпрограммой, можно использовать приведенный выше шаблон, но эта модель не является безопасной, если `Icon` вызывается параллельно. Как и исходная функция `Deposit`, `Icon` выполняется в несколько этапов: проверяет, не равно ли `nil` значение `icons`, загружает пиктограммы и обновляет значение `icons`. Интуитивно можно предположить, что наихудший возможный результат состояния гонки — то, что функция `loadIcons` вызывается несколько раз. Пока первая go-подпрограмма занята загрузкой пиктограмм, вторая go-подпрограмма входит в функцию `Icon`, обнаруживает, что значение переменной равно `nil`, и также вызывает `loadIcons`.

Но и здесь интуиция подводит. (Мы надеемся, что теперь вы выработаете новое интуитивное представление о параллелизме — что доверять интуиции относительно параллелизма нельзя!) Вспомните обсуждение памяти в разделе 9.4. В отсутствие явной синхронизации компилятор и процессор могут свободно переупорядочивать доступы

к памяти любыми способами при условии, что поведение каждой go-подпрограммы остается последовательно согласованным. Ниже приведено одно из возможных переупорядочений инструкций `loadIcons`. Оно сохраняет пустое отображение в переменной `icons` до его заполнения:

```
func loadIcons() {
    icons = make(map[string]image.Image)
    icons["spades.png"]   = loadIcon("spades.png")
    icons["hearts.png"]   = loadIcon("hearts.png")
    icons["diamonds.png"] = loadIcon("diamonds.png")
    icons["clubs.png"]    = loadIcon("clubs.png")
}
```

Следовательно, go-подпрограмма, обнаружившая, что значение `icons` не равно `nil`, не может считать, что инициализация этой переменной выполнена полностью.

Простейший корректный способ гарантировать, что все go-подпрограммы увидят результат `loadIcons`, — синхронизировать их с применением мьютекса:

```
var mu sync.Mutex       // Защищает icons
var icons map[string]image.Image

// Безопасно с точки зрения параллельности.
func Icon(name string) image.Image {
    mu.Lock()
    defer mu.Unlock()
    if icons == nil {
        loadIcons()
    }
    return icons[name]
}
```

Однако ценой реализации взаимно исключающего доступа к `icons` является то, что две go-подпрограммы не смогут получить доступ к переменной одновременно даже после того, как переменная будет безопасно инициализирована и никогда не будет изменена снова. Это предполагает применение блокировки для нескольких читателей:

```
var mu sync.RWMutex     // Защищает icons
var icons map[string]image.Image

// Безопасно с точки зрения параллельности.
func Icon(name string) image.Image {
    mu.RLock()
    if icons != nil {
        icon := icons[name]
        mu.RUnlock()
        return icon
    }
    mu.RUnlock()

    // Исключительная блокировка
```

```
    mu.Lock()
    if icons == nil { // Примечание: нужна повторная проверка на nil
        loadIcons()
    }
    icon := icons[name]
    mu.Unlock()
    return icon
}
```

Теперь в наличии два критических раздела. Go-подпрограмма сначала выполняет блокировку для чтения, проверяет отображение, а затем снимает блокировку. Если запись найдена (основной случай), она возвращается функцией. Если запись не найдена, go-подпрограмма осуществляет блокировку для записи. Нет способа обновить неисключительную блокировку до исключительной без предварительного снятия блокировки, а потому мы должны перепроверить переменную `icons` на тот случай, если другая go-подпрограмма уже инициализировала ее в промежутке между блокировками.

Такой шаблон обеспечивает более высокую степень параллелизма, но является сложным и, таким образом, подверженным ошибкам. К счастью, пакет `sync` предоставляет специализированное средство решения проблемы однократной инициализации: `sync.Once`. Концептуально `Once` состоит из мьютекса и логической переменной, записывающей, имела ли инициализация место; мьютекс защищает как логическое значение, так и структуры данных клиента. Единственный метод `Do` принимает в качестве своего аргумента функцию инициализации. Давайте используем `Once` для упрощения функции `Icon`:

```
var loadIconsOnce sync.Once
var icons map[string]image.Image

// Безопасно с точки зрения параллельности.
func Icon(name string) image.Image {
    loadIconsOnce.Do(loadIcons)
    return icons[name]
}
```

Каждый вызов `Do(loadIcons)` блокирует мьютекс и выполняет проверку булевой переменной. В первом вызове, когда переменная имеет значение `false`, `Do` вызывает функцию `loadIcons`, а значение булевой переменной становится равным `true`. Последующие вызовы ничего не делают, но синхронизация с помощью мьютекса гарантирует, что влияние `loadIcons` на память (конкретно на `icons`) становится видимым для всех go-подпрограмм. Используя `sync.Once` таким образом, мы можем избежать совместного использования переменных другими go-подпрограммами до их корректного конструирования.

Упражнение 9.2. Перепишите пример `PopCount` из раздела 2.6.2 так, чтобы он инициализировал таблицу поиска с использованием `sync.Once` при первом к ней обращении. (В реальности стоимость синхронизации для таких малых и высокооптимизированных функций, как `PopCount`, является чрезмерно высокой.)

9.6. Детектор гонки

Даже при максимальной аккуратности все равно слишком легко допустить ошибки параллелизма. К счастью, среда выполнения Go и ее инструментарий оснащены интеллектуальным, но простым в использовании инструментом динамического анализа — *детектором гонки*.

Просто добавьте флаг `-race` к команде `go build`, `go run` или `go test`. Это заставит компилятор создать модифицированную версию вашего приложения или теста с дополнительным инструментарием, который эффективно записывает все обращения к совместно используемым переменным, произошедшим во время выполнения, наряду с информацией о go-подпрограмме, которая читает или записывает переменную. Кроме того, измененная программа записывает все события синхронизации, такие как инструкции `go`, операции над каналами, вызовы `(*sync.Mutex).Lock`, `(*sync.WaitGroup).Wait` и т.д. (Полный набор событий синхронизации определяется документом *The Go Memory Model* (Модель памяти Go), который содержится в спецификации языка.)

Детектор гонки изучает этот поток событий в поисках случаев, когда одна go-подпрограмма считывает или записывает совместно используемую переменную, которая совсем недавно была записана другой go-подпрограммой, без промежуточных операций синхронизации. Это означает одновременный доступ к такой переменной и, таким образом, гонку данных. Инструментарий выводит отчет, включающий идентификатор переменной, и стеки вызовов активных функций в go-подпрограммах чтения и записи. Этого обычно достаточно, чтобы точно локализовать проблему. В разделе 9.7 содержится пример детектора гонки в действии.

Детектор гонки сообщает обо всех гонках данных, которые фактически были выполнены. Однако он может обнаружить только те состояния гонки, которые происходят во время выполнения; он не может доказать, что никакие другие гонки не будут когда-либо происходить. Для достижения наилучших результатов убедитесь, что ваши тесты проверяют ваши пакеты с использованием параллелизма.

Из-за дополнительных действий по записи информации программа, построенная с обнаружением гонки, требует при работе больше памяти и работает дольше, но накладные расходы вполне приемлемы даже для многих производственных программ. Для редко встречающегося состояния гонки включение детектора может сэкономить часы или даже дни отладки.

9.7. Пример: параллельный неблокирующий кеш

В этом разделе мы будем создавать *параллельный неблокирующий кеш* — абстракцию, которая решает проблему, часто возникающую в реальных параллельных программах, но не решенную окончательно существующими библиотеками. Это проблема *функций с запоминанием*, т.е. кеширование результата функции, так что его нужно вычислять только один раз. Наше решение будет безопасно с точки зрения параллель-

ности и позволит избежать конфликтов, связанных с дизайном на основе единственной блокировки для всего кеша.

В качестве примера функции с запоминанием воспользуемся функцией `httpGetBody`, показанной ниже. Она делает запрос HTTP GET и читает тело ответа. Вызовы этой функции являются относительно дорогими, поэтому мы хотели бы избежать их излишнего повторения.

```
func httpGetBody(url string) (interface{}, error) {
    resp, err := http.Get(url)
    if err != nil {
        return nil, err
    }
    defer resp.Body.Close()
    return ioutil.ReadAll(resp.Body)
}
```

В последней строке скрывается небольшая тонкость. `ReadAll` возвращает два результата, `[]byte` и `error`, но поскольку эти типы присваиваемы объявленным результатам `httpGetBody` — `interface{}` и `error` соответственно, — мы можем просто вернуть результат вызова без дальнейших церемоний. Мы выбрали этот тип возвращаемого значения для `httpGetBody` так, чтобы он соответствовал типу функций, для которых разрабатывается наш кеш, предназначенный для запоминания.

Вот первый набросок кеша:

gopl.io/ch9/memo1

```
// Пакет memo обеспечивает безопасное с точки зрения
// параллельности запоминание функции типа Func.
package memo

// Memo кеширует результаты вызовов Func.
type Memo struct {
    f     Func
    cache map[string]result
}

// Func является типом функции с запоминанием.
type Func func(key string) (interface{}, error)
type result struct {
    value interface{}
    err   error
}

func New(f Func) *Memo {
    return &Memo{f: f, cache: make(map[string]result)}
}

// Примечание: небезопасно с точки зрения параллельности!
func (memo *Memo) Get(key string) (interface{}, error) {
```

```
    res, ok := memo.cache[key]
    if !ok {
        res.value, res.err = memo.f(key)
        memo.cache[key] = res
    }
    return res.value, res.err
}
```

Экземпляр `Memo` содержит функцию `f` (результаты выполнения которой будут запоминаться) с типом `Func` и кеш, который представляет собой отображение строк на `result`. Каждый `result` представляет собой просто пару результатов, возвращаемых вызовом `f`, — значение и ошибку. Мы покажем несколько вариаций `Memo` по ходу развития дизайна, но все они будут использовать эти базовые свойства.Пример использования `Memo` показан ниже. Для каждого элемента в потоке входящих URL мы вызываем `Get` и протоколируем продолжительность вызова и количество данных, которые он возвращает:

```
m := memo.New(httpGetBody)
for url := range incomingURLs() {
    start := time.Now()
    value, err := m.Get(url)
    if err != nil {
        log.Print(err)
    }
    fmt.Printf("%s, %s, %d байтов\n",
               url, time.Since(start), len(value.([]byte)))
}
```

Для систематического исследования эффекта запоминания можно использовать пакет `testing` (это тема главы 11, "Тестирование"). Из выходных данных теста, приведенных ниже, видно, что поток URL содержит дубликаты и что, хотя первый вызов `(*Memo).Get` для каждого URL занимает сотни миллисекунд, второй запрос возвращает такое же количество данных в пределах миллисекунды:

```
$ go test v
gopl.io/ch9/memo1
=== RUN Test
https://golang.org, 175.026418ms, 7537 байтов
https://godoc.org, 172.686825ms, 6878 байтов
https://play.golang.org, 115.762377ms, 5767 байтов
http://gopl.io, 749.887242ms, 2856 байтов
https://golang.org, 721ns, 7537 байтов
https://godoc.org, 152ns, 6878 байтов
https://play.golang.org, 205ns, 5767 байтов
http://gopl.io, 326ns, 2856 байтов
--- PASS: Test (1.21s)
PASS
ok gopl.io/ch9/memo1 1.257s
```

Этот тест выполняет все вызовы `Get` последовательно.

Поскольку HTTP-запросы — отличная возможность применения параллелизма, давайте изменим тест так, чтобы он выполнял все запросы параллельно. В тесте используется `sync.WaitGroup` — для ожидания при завершении программы, пока последний запрос не будет полностью выполнен:

```
m := memo.New(httpGetBody)
var n sync.WaitGroup
for url := range incomingURLs() {
    n.Add(1)
    go func(url string) {
        start := time.Now()
        value, err := m.Get(url)
        if err != nil {
            log.Print(err)
        }
        fmt.Printf("%s, %s, %d байтов\n",
                   url, time.Since(start), len(value.([]byte)))
        n.Done()
    }(url)
}
n.Wait()
```

Этот тест выполняется гораздо быстрее, но, к сожалению, непохоже, что он корректно работает все время. Мы можем заметить неожиданные отсутствия информации в кеше или выборки из кеша, возвращающие неправильные значения, или даже аварийные ситуации.

Но хуже всего то, что он может корректно работать *некоторое* время, так что мы можем даже не заметить наличие проблем. Но если мы запустим код с флагом `-race`, детектор гонки (раздел 9.6) может вывести отчет наподобие следующего:

```
$ go test -run=TestConcurrent -race -v gopl.io/ch9/memo1
=== RUN TestConcurrent
...
WARNING: DATA RACE
Write by goroutine 36:
    runtime.mapassign1()
        ~/go/src/runtime/hashmap.go:411 +0x0
    gopl.io/ch9/memo1.(*Memo).Get()
        ~/gobook2/src/gopl.io/ch9/memo1/memo.go:32 +0x205
    ...
Previous write by goroutine 35:
    runtime.mapassign1()
        ~/go/src/runtime/hashmap.go:411 +0x0
    gopl.io/ch9/memo1.(*Memo).Get()
        ~/gobook2/src/gopl.io/ch9/memo1/memo.go:32 +0x205
...
Found 1 data race(s)
FAIL gopl.io/ch9/memo1 2.393s
```

Ссылка на memo.go:32 говорит нам о том, что две go-подпрограммы обновили отображение кеша без какой-либо промежуточной синхронизации. Метод Get не является безопасным с точки зрения параллельности: в нем имеется гонка данных:

```
28 func (memo *Memo) Get(key string) (interface{}, error) {
29     res, ok := memo.cache[key]
30     if !ok {
31         res.value, res.err = memo.f(key)
32         memo.cache[key] = res
33     }
34     return res.value, res.err
35 }
```

Самый простой способ сделать кеш безопасным с точки зрения параллельности — использовать синхронизацию на основе монитора. Все, что нужно сделать, — это добавить мьютекс в Memo, захват мьютекса в начале Get и освобождение его до выхода из Get так, чтобы две операции с cache выполнялись в критическом разделе:

gopl.io/ch9/memo2

```
type Memo struct {
    f     Func
    mu    sync.Mutex   // Защита cache
    cache map[string]result
}

// Метод Get безопасен с точки зрения параллельности.
func (memo *Memo) Get(key string) (value interface{}, err error) {
    memo.mu.Lock()
    res, ok := memo.cache[key]
    if !ok {
        res.value, res.err = memo.f(key)
        memo.cache[key] = res
    }
    memo.mu.Unlock()
    return res.value, res.err
}
```

Теперь детектор гонки молчит даже при параллельном выполнении тестов. К сожалению, это изменение Memo отменяет добытый ранее прирост производительности. Применяя блокировку на время, равное продолжительности каждого вызова f, Get сериализует все операции ввода-вывода, которые мы намеревались сделать параллельными. Нам нужен *неблокирующий кеш*, не сериализующий вызовы функции, для запоминания результатов которой он предназначен.

В следующей реализации Get вызывающая go-подпрограмма выполняет блокировку дважды: один раз — для поиска, а второй раз — для обновления, если поиск неудачен. Между ними другие go-подпрограммы могут свободно использовать кеш.

gopl.io/ch9/memo3

```
func (memo *Memo) Get(key string) (value interface{}, err error) {
    memo.mu.Lock()
    res, ok := memo.cache[key]
    memo.mu.Unlock()
    if !ok {
        res.value, res.err = memo.f(key)

        // Между этими двумя критическими разделами
        // несколько go-подпрограмм могут вычислять
        // f(key) и обновлять отображение.
        memo.mu.Lock()
        memo.cache[key] = res
        memo.mu.Unlock()
    }
    return res.value, res.err
}
```

Производительность снова улучшается, но теперь мы замечаем, что некоторые URL выбираются дважды. Это происходит, когда две или более go-подпрограмм вызывают `Get` для одного и того же URL в одно и то же время. Обе выполняют поиск в кеше, не находят искомого значения, а затем вызывают медленную функцию `f`. После этого обе go-подпрограммы обновляют отображение полученным результатом. В итоге один результат заменяется другим.

В идеале хотелось бы избежать такой лишней работы. Эта возможность иногда называется *подавлением повторений*. В версии `Memo` ниже каждый элемент отображения является указателем на структуру `entry`. Каждый элемент `entry` содержит записанный результат вызова функции `f`, как и прежде, но теперь он дополнительно содержит канал с именем `ready`. Сразу после того, как поле `result` структуры `entry` оказывается записанным, этот канал закрывается, широковещательно оповещая (раздел 8.9) все прочие go-подпрограммы о том, что теперь можно безопасно читать результат из этого элемента `entry`.

gopl.io/ch9/memo4

```
type entry struct {
    res   result
    ready chan struct{} // Закрывается, когда res готов
}

func New(f Func) *Memo {
    return &Memo{f: f, cache: make(map[string]*entry)}
}

type Memo struct {
    f     Func
    mu    sync.Mutex    // Защита cache
    cache map[string]*entry
}
```

```
func (memo *Memo) Get(key string) (value interface{}, err error) {
    memo.mu.Lock()
    e := memo.cache[key]
    if e == nil {
        // Это первый запрос данного ключа.
        // Эта go-подпрограмма становится ответственной за
        // вычисление значения и оповещение о готовности.
        e = &entry{ready: make(chan struct{})}
        memo.cache[key] = e
        memo.mu.Unlock()

        e.res.value, e.res.err = memo.f(key)

        close(e.ready)  // Широковещательное оповещение о готовности
    } else {
        // Это повторный запрос данного ключа.
        memo.mu.Unlock()

        <-e.ready       // Ожидание готовности
    }
    return e.res.value, e.res.err
}
```

Вызов `Get` теперь включает в себя захват мьютекса, охраняющего отображение кеша, поиск в отображении указателя на существующую запись `entry`, выделение памяти и вставку новой записи, если поиск был неудачен, и освобождение мьютекса. Если же имеется существующая запись, ее значение не обязательно готово — в этот момент другая go-подпрограмма может осуществлять вызов медленной функции `f`, — так что вызывающая go-подпрограмма должна ожидать оповещения о готовности, прежде чем читать `result` из записи `entry`. Она делает это, выполняя чтение из канала `ready`, поскольку эта операция блокирует go-подпрограмму до тех пор, пока канал не будет закрыт.

Если нет соответствующего существующего элемента `entry`, то путем вставки нового "не готового" элемента в отображение текущая go-подпрограмма берет на себя ответственность за вызов медленной функции, обновление записи и оповещение о готовности новой записи всех других go-подпрограмм, которые могут (к тому времени) ее ожидать.

Обратите внимание, что переменные `e.res.value` и `e.res.err` в `entry` совместно используются несколькими go-подпрограммами. Go-подпрограмма, которая создает `entry`, устанавливает их значения, а другие go-подпрограммы читают их значения только после широковещательного оповещения о состоянии готовности. Несмотря на обращение со стороны нескольких go-подпрограмм, мьютекс не является необходимым. Закрытие канала `ready` *предшествует* получению оповещения любыми другими go-подпрограммами, поэтому запись в эти переменные в первой go-подпрограмме *предшествует* их чтению последующими go-подпрограммами. В результате никакой гонки данных не существует.

Наш параллельный, неблокирующий кеш без дублирования готов.

Показанная выше реализация `Memo` использует мьютекс для защиты переменной отображения, которая используется совместно всеми go-подпрограммами, которые вызывают метод `Get`. Интересно сравнить этот дизайн с альтернативным, в котором переменная отображения ограничена управляющей go-подпрограммой, которой абоненты `Get` должны отправлять сообщение.

Объявления `Func`, `result` и `entry` остаются такими же, как и ранее:

```
// Func представляет собой тип функции,
// для которой реализуется запоминание.
type Func func(key string) (interface{}, error)
// result представляет собой результат вызова Func.
type result struct {
    value interface{}
    err   error
}
type entry struct {
    res   result
    ready chan struct{} // Закрыт, когда res готов
}
```

Однако тип `Memo` теперь состоит из канала, `requests`, через который функция, вызывающая `Get`, взаимодействует с управляющей go-подпрограммой. Типом элемента канала является `request`. С помощью этой структуры вызывающая `Get` функция отправляет управляющей go-подпрограмме как ключ, т.е. аргумент функции с запоминанием, так и еще один канал, `response`, по которому результат должен отправляться обратно, когда он становится доступным. Этот канал будет переносить только одно значение.

gopl.io/ch9/memo5

```
// request представляет собой сообщение,
// требующее применения Func к key.
type request struct {
    key      string
    response chan<- result
    // Клиенту нужен только result
}

type Memo struct{ requests chan request }

// New возвращает f с запоминанием.
// Впоследствии клиенты должны вызывать Close.
func New(f Func) *Memo {
    memo := &Memo{requests: make(chan request)}
    go memo.server(f)
    return memo
}
```

```
func (memo *Memo) Get(key string) (interface{}, error) {
    response := make(chan result)
    memo.requests <- request{key, response}
    res := <-response
    return res.value, res.err
}
func (memo *Memo) Close() { close(memo.requests) }
```

Приведенный выше метод `Get` создает канал `response`, помещает его в запрос и отправляет запрос управляющей go-подпрограмме, после чего сразу же переходит к получению ответа из этого канала.

Переменная `cache` ограничена управляющей go-подпрограммой `(*Memo).server`, показанной ниже. Она считывает запросы в цикле до тех пор, пока канал не будет закрыт с помощью метода `Close`. Для каждого запроса она обращается к кешу, создавая и вставляя новую запись `entry`, если таковая не была найдена.

```
func (memo *Memo) server(f Func) {
    cache := make(map[string]*entry)
    for req := range memo.requests {
        e := cache[req.key]
        if e == nil {
            // Это первый запрос данного ключа key.
            e = &entry{ready: make(chan struct{})}
            cache[req.key] = e
            go e.call(f, req.key) // Вызов f(key)
        }
        go e.deliver(req.response)
    }
}
func (e *entry) call(f Func, key string) {
    // Вычисление функции.
    e.res.value, e.res.err = f(key)
    // Оповещение о готовности.
    close(e.ready)
}
func (e *entry) deliver(response chan<- result) {
    // Ожидание готовности.
    <-e.ready
    // Отправка результата клиенту.
    response <- e.res
}
```

Способом, аналогичным версии на основе мьютекса, первый запрос некоторого ключа становится ответственным за вызов функции `f` для этого ключа, сохранение результата в `entry` и оповещение о готовности с помощью закрытия `ready`. Это делается с помощью вызова `(*entry).call`.

Последующий запрос того же ключа находит существующую запись `entry` в отображении, ждет, когда результат будет готов, и отправляет его по каналу клиентской

go-подпрограмме, которая вызвала `Get`. Это делается с помощью вызова `(*entry).deliver`. Методы `call` и `deliver` должны вызываться в их собственных go-подпрограммах, чтобы гарантировать, что управляющая go-подпрограмма не остановит обработку новых запросов.

Этот пример показывает, что можно построить много параллельных структур, используя любой из подходов — с совместно используемыми переменными и блокировками или со взаимодействующими последовательными процессами, — без чрезмерной сложности.

Не всегда очевидно, какой именно подход предпочтителен в данной ситуации, но стоит знать оба. Иногда переход от одного подхода к другому позволяет существенно упростить код.

Упражнение 9.3. Расширьте тип `Func` и метод `(*Memo).Get` так, чтобы вызывающая функция могла предоставить необязательный канал `done`, с помощью которого можно было бы отменить операцию (раздел 8.9). Результаты отмененного вызова `Func` кешироваться не должны.

9.8. Go-подпрограммы и потоки

В предыдущей главе мы говорили, что пока можно игнорировать разницу между go-подпрограммами и потоками операционной системы (ОС). Хотя различия между ними, по существу, количественные, достаточно большая количественная разница становится качественной, и это справедливо и для go-подпрограмм и потоков. Теперь настало время посмотреть на их различия.

9.8.1. Растущие стеки

Каждый поток операционной системы имеет блок памяти фиксированного размера (зачастую до 2 Мбайт) для *стека* — рабочей области, в которой он хранит локальные переменные вызовов функций, находящиеся в работе или приостановленные на время вызова другой функции. Такой стек фиксированного размера является одновременно слишком большим и слишком малым. Стек размером 2 Мбайта оказывается огромной тратой памяти для небольшой go-подпрограммы, которая, например, просто ожидает `WaitGroup`, а затем закрывает канал. Для программ Go не редкость одновременное создание сотен тысяч go-подпрограмм, что было бы невозможно с такими большими стеками. Тем не менее стеков фиксированного размера, несмотря на их большие размеры, не всегда достаточно для сложных и глубоко рекурсивных функций. Изменение фиксированного размера стека может повысить эффективность использования памяти и позволить создавать большее количество потоков или разрешить включать рекурсивные функции с большей глубиной, но не может одновременно обеспечить и то, и другое.

В противоположность этому go-подпрограмма начинает работу с небольшим стеком, обычно около 2 Кбайт. Стек go-подпрограммы, подобно стеку потока операционной системы, хранит локальные переменные активных и приостановленных функций, но, в отличие от потоков операционной системы, не является фиксирован-

ным; при необходимости он может расти и уменьшаться. Максимальный размер стека go-подпрограммы может быть около 1 Гбайта, на порядки больше типичного стека с фиксированным размером, хотя, конечно, такой большой стек могут использовать только несколько go-подпрограмм.

Упражнение 9.4. Постройте конвейер, соединяющий произвольное количество go-подпрограмм каналами. Каково максимальное количество этапов конвейера, который можно создать без исчерпания памяти? Сколько времени длится транзит значения через весь конвейер?

9.8.2. Планирование go-подпрограмм

Потоки операционной системы планируются в ее ядре. Каждые несколько миллисекунд аппаратный таймер прерывает процессор, что приводит к вызову функции ядра, именуемой *планировщиком*. Эта функция приостанавливает работу текущего потока и сохраняет значения его регистров в памяти, просматривает список потоков и решает, какой из них следует запустить следующим, восстанавливает регистры этого потока из памяти и возобновляет его выполнение. Так как потоки операционной системы планируются в ядре, передача управления от одного потока к другому требует полного *переключения контекста*, т.е. сохранения состояния одного пользовательского потока в памяти, восстановление состояния другого и обновление структур данных планировщика. Это медленная операция — из-за слабой локальности и необходимого количества обращений к памяти.

Среда выполнения Go содержит собственный планировщик, который использует метод, известный как *m:n-планирование*, потому что он мультиплексирует (или планирует) выполнение *m* go-подпрограмм на *n* потоках операционной системы. Задания планировщика Go аналогичны заданиям планировщика ядра, но связаны только с go-подпрограммами одной программы Go.

В отличие от планировщика потоков операционной системы планировщик Go вызывается не периодически аппаратным таймером, а неявно некоторыми конструкциями языка Go. Например, когда go-подпрограмма вызывает `time.Sleep` или блокируется операцией канала или мьютекса, планировщик переводит ее в спящий режим и запускает другую go-подпрограмму до тех пор, пока не наступит время активировать первую. Поскольку переключение контекста ядра не требуется, планирование go-подпрограмм намного дешевле, чем потоков.

Упражнение 9.5. Напишите программу с двумя go-подпрограммами, которая отправляет сообщения назад и вперед по двум небуферизованным каналам наподобие мячика для пинг-понга. Какое количество сообщений в секунду может поддерживать эта программа?

9.8.3. GOMAXPROCS

Планировщик Go использует параметр с именем `GOMAXPROCS` для определения, сколько потоков операционной системы могут одновременно активно выполнять код Go. Его значение по умолчанию равно количеству процессоров компьютера, так что

на машине с 8 процессорами планировщик будет планировать код Go для выполнения на 8 потоках одновременно (GOMAXPROCS равно значению *n* в *m:n-планировании*). Спящие или заблокированные в процессе коммуникации go-подпрограммы потоков для себя не требуют. Go-подпрограммы, заблокированные в операции ввода-вывода или в других системных вызовах, или при вызове функций, не являющихся функциями Go, нуждаются в потоке операционной системы, но GOMAXPROCS их не учитывает.

Можно явно управлять этим параметром с помощью переменной среды GOMAXPROCS или функции `runtime.GOMAXPROCS`. Мы можем увидеть влияние GOMAXPROCS на следующем маленьком примере программы, которая выводит бесконечный поток нулей и единиц:

```
for {
    go fmt.Print(0)
    fmt.Print(1)
}
```

```
$ GOMAXPROCS=1 go run hacker-cliché.go
111111111111111111100000000000000000000011111...

$ GOMAXPROCS=2 go run hacker-cliché.go
010101010101010101011001100101011010010100110...
```

При первом запуске одновременно выполняется не более одной go-подпрограммы. Первоначально это главная go-подпрограмма, выводящая единицы. Спустя некоторое время планировщик приостанавливает ее и активирует go-подпрограмму, выводящую нули, выделяя ей время для работы в потоке операционной системы. При втором запуске имеются два потока операционной системы, так что обе go-подпрограммы работают одновременно, выводя цифры примерно с одной и той же скоростью. Мы должны подчеркнуть, что в планировании go-подпрограмм участвуют многие факторы, а среда выполнения постоянно развивается, так что ваши результаты могут отличаться от приведенных выше.

Упражнение 9.6. Измерьте влияние значения GOMAXPROCS на производительность параллельной программы из упражнения 8.5. Каково оптимальное значение на вашем компьютере? Сколько процессоров есть в вашем компьютере?

9.8.4. Go-подпрограммы не имеют идентификации

В большинстве операционных систем и языков программирования, поддерживающих многопоточность, текущий поток имеет идентификацию, которая может быть легко получена как обычное значение (обычно — целое число или указатель). Это облегчает построение абстракции, именуемой *локальной памятью потока*, которая, по существу, является глобальным отображением, использующим в качестве ключа идентификатор потока, так что каждый поток может сохранять и извлекать значения независимо от других потоков.

У go-подпрограмм нет понятия идентификации, доступной программисту. Так решено во время проектирования языка, поскольку локальной памятью потока про-

граммисты, как правило, злоупотребляют. Например, на веб-сервере, реализованном на языке с локальной памятью потока, многие функции часто ищут в этой памяти информацию об HTTP-запросе, от имени которого они работают в настоящее время. Однако так же, как и в случае программ, чрезмерно использующих глобальные переменные, это может привести к нездоровым "связям на расстоянии", когда поведение функции определяется не только аргументами, но и идентификатором потока, в котором она выполняется. Таким образом, при необходимости изменить идентификатор потока такая функция поведет себя некорректно, причем обнаружить причину такого поведения будет крайне сложно.

Go поощряет простой стиль программирования, при котором все параметры, влияющие на поведение функции, являются явными. Это не только делает программы более удобочитаемыми, но и позволяет нам свободно распределять подзадачи данной функции по многим разным go-подпрограммам, не беспокоясь об их идентификации.

Итак, теперь вы узнали обо всех возможностях языка, которые нужны для написания программ на Go. В следующих двух главах мы вернемся немного назад, чтобы взглянуть на некоторые практические методы и инструменты, поддерживающие крупномасштабное программирование: как структурировать проект как набор пакетов, как получать, создавать, тестировать, профилировать и документировать пакеты, а также как делиться ими с другими программистами.

10

Пакеты и инструменты Go

Сегодня программа скромного размера может содержать десятки тысяч функций. Тем не менее ее автору нужно думать лишь о некоторых из них, а разрабатывать — и того меньше, потому что подавляющее большинство функций написано другими программистами и доступно для повторного использования в виде *пакетов*.

Go поставляется более чем с сотней стандартных пакетов, которые обеспечивают основу для большинства приложений. Куда больше пакетов, разработанных всем сообществом программистов Go, можно найти по адресу `http://godoc.org`. В этой главе мы покажем, как использовать существующие пакеты и создавать новые.

Go также поставляется инструментом `go`, сложным, но простым в использовании, который представляет собой команду в том числе для управления пакетами Go. В начале книги мы показали, как использовать `go` для того, чтобы скачать, построить и запустить примеры программ. В этой главе рассмотрены основные концепции, лежащие в основе этого инструмента, и подробно изложены его возможности, которые включают вывод документации и запрос метаданных о пакетах в рабочей области. В следующей главе мы будем изучать его возможности тестирования.

10.1. Введение

Цель любой системы пакетов — сделать дизайн и поддержку больших программ практичными путем группирования связанных функций в модули, которые легко понимать и изменять независимо от других пакетов программы. Такая *модульность* позволяет совместно использовать пакеты разными проектами, распространять их в пределах организации или делать доступными всему миру.

Каждый пакет определяет уникальное пространство имен, охватывающее все его идентификаторы. Каждое имя связано с конкретным пакетом, что позволяет нам выбирать короткие, ясные имена для наиболее часто используемых типов, функций и так далее, не создавая при этом конфликтов с другими частями программы.

Пакеты также обеспечивают *инкапсуляцию*, управляя тем, какие имена видны, или экспортируемы, вне пакета. Ограничение видимости членов пакета скрывает вспомогательные функции и типы за API пакета, позволяя сопровождению пакета менять внутреннюю реализацию, не затрагивая код вне пакета. Ограничение видимости также скрывает переменные, так что клиенты могут получать к ним доступ и обновлять

их значения только через экспортированные функции, которые обеспечивают сохранение внутренних инвариантов или взаимоисключение в параллельной программе.

При изменении файла необходимо перекомпилировать пакет, в который входит этот файл, и потенциально — все пакеты, которые зависят от него. Компиляция Go выполняется гораздо быстрее, чем большинства других компилируемых языков, даже когда построение выполняется "с нуля". Есть три основные причины такой скорости компиляции. Во-первых, все импортируемые пакеты должны быть явно указаны в начале каждого исходного файла, поэтому компилятору не приходится читать и анализировать весь файл, чтобы определить его зависимости. Во-вторых, зависимости пакета образуют ориентированный ациклический граф, и в силу отсутствия циклов пакеты могут компилироваться отдельно и, возможно, параллельно. Наконец, в-третьих, объектный файл скомпилированного пакета Go записывает экспортируемую информацию не только для самого пакета, но и для его зависимостей. При компиляции пакета компилятор должен прочитать один объектный файл для каждого импортируемого пакета, но не должен выходить за пределы этих файлов.

10.2. Пути импорта

Каждый пакет идентифицируется уникальной строкой, которая называется его *путем импорта*. Пути импорта являются строками, которые участвуют в объявлениях `import`.

```
import (
    "fmt"
    "math/rand"
    "encoding/json"
    "golang.org/x/net/html"
    "github.com/gosqldriver/
    mysql"
)
```

Как упоминалось в разделе 2.6.1, спецификация языка Go не определяет смысл этих строк или то, как находить пути импорта пакета, оставляя эти вопросы инструментарию. В этой главе мы подробно рассмотрим, как их интерпретирует инструмент `go`, поскольку именно его большинство программистов Go используют для создания, тестирования и т.д. Тем не менее существуют и другие инструменты. Например, программисты Go, использующие внутреннюю многоязычную систему построения Google, следуют иным правилам именования и определения местонахождения пакетов, указания тестов и тому подобного, более точно соответствующим соглашениям этой системы.

Для пакетов, которые предназначены для совместного использования или публикации, пути импорта должны быть глобально уникальными. Чтобы избежать конфликтов, пути импорта всех пакетов, не входящих в стандартную библиотеку, должны начинаться с доменного имени организации, которая владеет пакетом; это также позволяет легко находить пакеты. Например, приведенное выше объявление импор-

тирует HTML-анализатор, поддерживаемый командой Go, и популярный драйвер базы данных MySQL стороннего разработчика.

10.3. Объявление пакета

Объявление `package` требуется в начале каждого исходного файла Go. Его главная цель — определить идентификатор по умолчанию для этого пакета (называемый *именем пакета*) при его импорте в другой пакет.

Например, каждый файл пакета `math/rand` начинается с `package rand`, так что, импортируя этот пакет, вы можете обращаться к его членам как к `rand.Int`, `rand.Float64` и т.д.

```
package main
import (
    "fmt"
    "math/rand"
)

func main() {
    fmt.Println(rand.Int())
}
```

По соглашению имя пакета представляет собой последнюю часть пути импорта, и в результате два пакета могут иметь одно и то же имя, хотя их пути импорта различны. Например, пакеты с путями импорта `math/rand` и `crypto/rand` имеют имя `rand`. Далее мы увидим, как одновременно использовать оба пакета в одной и той же программе.

Существует три основных исключения из соглашения о последней части. Во-первых, пакет, определяющий команду (выполнимую программу Go), всегда имеет имя `main` независимо от пути импорта пакета. Это сигнал для `go build` (раздел 10.7.3) о том, что он должен вызывать компоновщик для создания выполнимого файла.

Во-вторых, некоторые файлы в каталоге могут иметь суффикс `_test` в имени пакета, если имя файла заканчивается на `_test.go`. Такой каталог может определять *два* пакета: обычный и еще один, именуемый *пакетом внешнего тестирования*. Суффикс `_test` сигнализирует `go test` о том, что надо построить оба пакета, и указывает, какие файлы принадлежат к каждому из пакетов. Пакеты внешнего тестирования используются, чтобы избежать циклов в графе импорта, получающихся из зависимостей теста; более подробно этот вопрос рассматривается в разделе 11.2.4.

В-третьих, некоторые инструменты для управления зависимостями добавляют суффикс с номером версии в путь импорта пакета, такой как `gopkg.in/yaml.v2`. Суффикс из имени пакета исключается, так что в данном случае имя пакета — просто `yaml`.

10.4. Объявления импорта

Исходный файл Go может содержать нуль или более объявлений `import` сразу после объявления `package` и перед первым объявлением, не являющимся объявлением импорта. Каждое объявление импорта может указывать путь импорта одного пакета или нескольких пакетов в списке в круглых скобках. Две показанные ниже формы эквивалентны, но вторая более распространена:

```
import "fmt"
import "os"

import (
    "fmt"
    "os"
)
```

Импортированные пакеты могут быть сгруппированы с помощью добавления пустых строк; такие группы обычно указывают различные предметные области. Порядок не важен, но по соглашению строки каждой группы сортируются в алфавитном порядке.

```
import (
    "fmt"
    "html/template"
    "os"

    "golang.org/x/net/html"
    "golang.org/x/net/ipv4"
)
```

Если нужно импортировать два пакета с одним и тем же именем, как, например, `math/rand` и `crypto/rand`, в третий пакет, объявление импорта должно указать альтернативное имя по крайней мере для одного из них, чтобы избежать конфликта. Это действие называется *переименованием импорта*.

```
import (
    "crypto/rand"
    mrand "math/rand" // Альтернативное имя mrand устраняет конфликт
)
```

Альтернативное имя затрагивает только импортирующий файл. Другие файлы, даже в том же пакете, могут импортировать пакет, используя его имя по умолчанию или другое имя.

Переименование импорта может быть полезным даже тогда, когда нет никакого конфликта. Если имя импортируемого пакета громоздкое, как это иногда бывает в случае автоматически созданного кода, сокращенное имя может оказаться более удобным. Во избежание путаницы следует последовательно использовать одно и то же короткое имя. Выбор альтернативного имени может помочь избежать конфликтов с распространенными именами локальных переменных. Например, в файле с многими

локальными переменными с именем path мы могли бы импортировать стандартный пакет "path" как pathpkg.

Каждое объявление импорта устанавливает зависимость текущего пакета от импортированного. Инструмент go build выдает сообщение об ошибке, если эти зависимости образуют цикл.

10.5. Пустой импорт

Нельзя импортировать пакет в файл, но при этом нигде в этом файле не сослаться на имя, которое он определяет. Это рассматривается как ошибка. Однако иногда необходимо импортировать пакет просто для побочных эффектов — вычисления инициализирующих выражений его переменных уровня пакета и выполнения функций init (раздел 2.6.2). Чтобы подавить ошибку "неиспользуемый импорт", с которой мы в противном случае столкнемся, следует использовать переименование импорта с альтернативным названием _ (пустой идентификатор). Как обычно, на пустой идентификатор можно никогда не ссылаться.

```
import _ "image/png" // Регистрация декодировщика PNG
```

Это действие известно как *пустой импорт*. Наиболее часто пустой импорт используется для реализации механизма времени компиляции, согласно которому основная программа может включать необязательные возможности с помощью пустого импорта дополнительных пакетов. Сначала мы посмотрим, как его использовать, а затем — как он работает.

Пакет стандартной библиотеки image экспортирует функцию Decode, которая считывает байты из io.Reader, определяет, какой формат изображения использован для кодирования данных, вызывает соответствующий декодер и возвращает результирующее значение image.Image. Используя image.Decode, легко создать простой преобразователь изображений, который считывает изображение в одном формате и записывает в другом:

gopl.io/ch10/jpeg

```
// Команда jpeg читает изображение PNG из стандартного ввода
// и записывает его как изображение JPEG на стандартный вывод.
package main

import (
    "fmt"
    "image"
    "image/jpeg"
    _ "image/png" // Регистрация PNG-декодера
    "io"
    "os"
)

func main() {
```

```
    if err := toJPEG(os.Stdin, os.Stdout); err != nil {
        fmt.Fprintf(os.Stderr, "jpeg: %v\n", err)
        os.Exit(1)
    }
}

func toJPEG(in io.Reader, out io.Writer) error {
    img, kind, err := image.Decode(in)
    if err != nil {
        return err
    }
    fmt.Fprintln(os.Stderr, "Входной формат =", kind)
    return jpeg.Encode(out, img, &jpeg.Options{Quality: 95})
}
```

Если мы передадим выход программы `gopl.io/ch3/mandelbrot` (раздел 3.3) программе преобразования, она обнаружит входной формат PNG и запишет JPEG-версию рис. 3.3.

```
$ go build gopl.io/ch3/mandelbrot
$ go build gopl.io/ch10/jpeg
$ ./mandelbrot | ./jpeg > mandelbrot.jpg
Входной формат = png
```

Обратите внимание на пустой импорт `image/png`. Без этой строки программа будет компилироваться и компоноваться как обычно, но может не распознавать или декодировать ввод в формате PNG:

```
$ go build gopl.io/ch10/jpeg
$ ./mandelbrot | ./jpeg >mandelbrot.jpg
jpeg: image: неизвестный формат
```

Вот как это работает. Стандартная библиотека предоставляет декодеры для GIF, PNG и JPEG, а пользователи могут предоставлять и другие. Но чтобы выполнимые файлы были небольшими, декодеры не включаются в приложение, если это явно не потребовать. Функция `image.Decode` консультируется с таблицей поддерживаемых форматов. Каждая запись в таблице указывает четыре вещи: имя формата; строку, которая представляет собой префикс всех изображений, закодированных таким образом, и используется для определения кодировки; функцию `Decode`, которая декодирует изображения; и еще одну функцию `DecodeConfig`, которая декодирует только метаданные изображения, например его размер и цветовое пространство. Запись добавляется в таблицу путем вызова `image.RegisterFormat`, обычно из инициализатора пакета поддержки каждого формата, как в пакете `image/png`:

```
package png        // image/png

func Decode(r io.Reader) (image.Image, error)
func DecodeConfig(r io.Reader) (image.Config, error)
```

```
func init() {
    const pngHeader = "\x89PNG\r\n\x1a\n"
    image.RegisterFormat("png", pngHeader, Decode, DecodeConfig)
}
```

Чтобы функция `image.Decode` была в состоянии декодировать данный формат, приложению требуется только пустой импорт пакета для регистрации формата.

Пакет `database/sql` использует аналогичный механизм для того, чтобы позволить пользователям установить только те драйверы баз данных, в которых они нуждаются, например:

```
import (
    "database/sql"
    _ "github.com/lib/pq" // enable support for Postgres
    _ "github.com/gosqldriver/
    mysql" // enable support for MySQL
)

db, err = sql.Open("postgres", dbname) // OK
db, err = sql.Open("mysql", dbname)    // OK
db, err = sql.Open("sqlite3", dbname)  // Возврат ошибки:
                                       // неизвестный драйвер "sqlite3"
```

Упражнение 10.1. Расширьте программу `jpeg` так, чтобы она преобразовывала любой поддерживаемый входной формат в любой выходной с использованием функции `image.Decode` для определения входного формата и флага командной строки для выбора выходного формата.

Упражнение 10.2. Определите обобщенную функцию чтения архива, способную читать ZIP-файлы (`archive/zip`) и POSIX tar-файлы (`archive/tar`). Воспользуйтесь механизмом регистрации, аналогичным описанному выше, чтобы поддержка каждого формата файла могла быть добавлена с помощью пустого импорта.

10.6. Пакеты и именование

В этом разделе мы предложим несколько советов о том, как следовать соглашениям Go для именования пакетов и их членов.

При создании пакета его имя следует делать коротким, но не настолько, чтобы оно стало непонятным. Наиболее часто используемые пакеты стандартной библиотеки имеют имена `bufio`, `bytes`, `flag`, `fmt`, `http`, `io`, `json`, `os`, `sort`, `sync` и `time`.

Где это возможно, используйте описательные и недвусмысленные имена. Например, не именуйте пакет утилит как `util`, при том что более конкретное имя `imageutil` или `ioutil` является вполне кратким. Не выбирайте имена пакетов, которые обычно используются для локальных переменных или могут заставить клиентов воспользоваться переименованием импорта, как в случае пакета `path`.

Обычно имена пакетов являются словами в единственном числе. Стандартные пакеты `bytes`, `errors` и `strings` используют множественное число, чтобы избежать

сокрытия соответствующих предопределенных типов и, в случае `go/types`, избежать конфликта с ключевым словом.

Избегайте имен пакетов, которые уже имеют другой смысл. Например, первоначально мы использовали для пакета преобразования температуры в разделе 2.5 имя `temp`, но это была ужасная идея, потому что "temp" является почти что синонимом для слова "временный". Какое-то время мы использовали имя `temperature`, но оно было слишком длинным и не поясняло, что именно делает пакет. В конце концов пакет получил имя `tempconv`, которое короче и похоже на `strconv`.

Теперь давайте обратимся к именованию членов пакета. Поскольку каждая ссылка на член другого пакета использует квалифицированный идентификатор, такой как `fmt.Println`, бремя описания члена пакета в равной мере возложено как на имя пакета, так и на имя члена. Мы не должны упоминать концепцию форматирования в имени `Println`, потому что это уже делает имя пакета `fmt`. При проектировании пакета рассмотрите, как работают вместе обе части квалифицированного идентификатора, а не только имя члена. Вот несколько характерных примеров:

```
bytes.Equal flag.Int http.Get json.Marshal
```

Мы можем выявить некоторые распространенные шаблоны именования. Пакет `strings` предоставляет ряд независимых функций для работы со строками:

```
package strings

func Index(needle, haystack string) int

type Replacer struct{ /* ... */ }
func NewReplacer(oldnew ...string) *Replacer

type Reader struct{ /* ... */ }
func NewReader(s string) *Reader
```

Слово `string` в любом из этих имен отсутствует. Клиенты обращаются к ним как к `strings.Index`, `strings.Replacer` и т.д.

Другие пакеты, которые мы могли бы описать как *пакеты одного типа*, такие как `html/template` и `math/rand`, предоставляют один главный тип данных плюс его методы и зачастую функцию `New` для создания его экземпляров.

```
package rand // "math/rand"

type Rand struct{ /* ... */ }
func New(source Source) *Rand
```

Это может привести к повторению, такому как `template.Template` или `rand.Rand`; именно поэтому имена пакетов такого рода часто являются особенно короткими.

С другой стороны есть такие пакеты, как `net/http`, которые имеют много имен без особой структуры, поскольку выполняют сложные задачи. Несмотря на наличие более 20 типов и гораздо большего количества функций, наиболее важные члены пакета имеют простейшие имена: `Get`, `Post`, `Handle`, `Error`, `Client`, `Server`.

10.7. Инструментарий Go

Остальная часть этой главы посвящена инструменту go, который используется для загрузки, запросов, форматирования, построения, тестирования и установки пакетов кода Go.

Инструмент go сочетает в себе возможности набора разнообразных инструментов в одной команде. Это менеджер пакетов (аналогично apt или rpm), который отвечает на запросы о перечне пакетов, вычисляет их зависимости и загружает их из удаленной системы управления версиями. Это система построения, которая вычисляет зависимости файлов и вызывает компиляторы, ассемблеры и компоновщики, хотя она преднамеренно менее полная, чем стандартная команда Unix make. И одновременно это тестировщик, как мы увидим в главе 11, "Тестирование".

Интерфейс его командной строки использует стиль "складного ножа", более чем с десятком подкоманд (с некоторыми из них мы уже знакомились), такими как get, run, build и fmt. Вы можете выполнить go help, чтобы увидеть предметный указатель встроенной документации, но для справки мы перечислим здесь наиболее часто используемые команды (в переводе):

```
$ go
...
    build   компиляция пакетов и зависимостей
    clean   удаление объектных файлов
    doc     документация по пакетам и именам
    env     вывод информации о среде Go
    fmt     запуск gofmt для исходных текстов пакета
    get     загрузка и установка пакетов и зависимостей
    install компиляция и установка пакетов и зависимостей
    list    список пакетов
    run     компиляция и выполнение програмы Go
    test    тестирование пакетов
    version вывод версии Go

Для получения более детальной информации о команде
используйте "go help [command]".
...
```

Чтобы сохранить минимальную потребность в конфигурации, инструмент go опирается на соглашения. Например, для данного имени файла исходного кода Go инструмент go может найти включающий его пакет, потому что каждый каталог содержит один пакет, и путь импорта пакета соответствует иерархии каталогов в рабочей области. Для данного пути импорта пакета инструмент может найти соответствующий каталог, в котором хранятся объектные файлы. Он может также найти URL сервера, на котором находится репозиторий исходного кода.

10.7.1. Организация рабочего пространства

Единственная настройка, которая требуется большинству пользователей, — это переменная среды GOPATH, которая определяет корневой каталог рабочей области. При переключении на другую рабочую область пользователи обновляют значение GOPATH. Например, при работе над этой книгой мы устанавливали GOPATH равным $HOME/gobook:

```
$ export GOPATH=$HOME/gobook
$ go get gopl.io/...
```

После загрузки всех программ для этой книги с помощью указанной выше команды ваша рабочая область будет содержать иерархию наподобие следующей:

```
GOPATH/
    src/
        gopl.io/
            .git/
            ch1/
                helloworld/
                    main.go
                dup/
                    main.go
                ...
        golang.org/x/net/
            .git/
            html/
                parse.go
                node.go
                ...
    bin/
        helloworld
        dup
    pkg/
        darwin_amd64/
            ...
```

GOPATH имеет три подкаталога. Подкаталоге src содержит исходный код. Каждый пакет находится в каталоге, имя которого по отношению к $GOPATH/src представляет собой путь импорта пакета, например gopl.io/ch1/helloworld. Заметим, что одна рабочая область GOPATH включает несколько репозиториев управления версиями в каталоге src, например gopl.io и golang.org. В подкаталоге pkg средства построения хранят скомпилированные пакеты, а подкаталог bin содержит выполнимые программы, такие как helloworld.

Вторая переменная среды, GOROOT, указывает корневой каталог дистрибутива Go, который предоставляет все пакеты стандартной библиотеки. Структура каталогов в GOROOT напоминает таковую в GOPATH, так что, например, исходные файлы пакета fmt располагаются в каталоге $GOROOT/src/fmt. Пользователям не нужно устанав-

ливать GOROOT, поскольку по умолчанию инструмент go будет использовать местоположение, в которое он был установлен.

Команда go env выводит действующие значения переменных среды, имеющих отношение к инструментарию, включая значения по умолчанию для отсутствующих. GOOS определяет целевую операционную систему (например, android, linux, darwin или windows), а GOARCH определяет архитектуру целевого процессора, например amd64, 386 или arm. Хотя единственной переменной, которую вы должны установить, является GOPATH, в наших объяснениях иногда появляются и другие переменные среды.

```
$ go env
GOPATH="/home/gopher/gobook"
GOROOT="/usr/local/go"
GOARCH="amd64"
GOOS="darwin"
...
```

10.7.2. Загрузка пакетов

При использовании инструмента go путь импорта пакета указывает не только, где его найти в локальной рабочей области, но и где его найти в Интернете, так что go get позволяет его получить и обновить.

Команда go get может загрузить один пакет или все поддерево или репозиторий с помощью записи ..., как в предыдущем разделе. Инструмент go также вычисляет и загружает все зависимости исходных пакетов; именно поэтому в рабочей области в предыдущем примере появился пакет golang.org/x/net/html.

После того как go get загрузит пакеты, он их строит, а затем *устанавливает* библиотеки и команды. Детальнее мы рассмотрим этот вопрос в следующем разделе, но на приведенном далее примере видно, насколько простой этот процесс. Первая приведенная ниже команда получает инструмент golint, который проверяет наличие распространенных проблем стиля в исходном коде Go. Вторая команда запускает golint для gopl.io/ch2/popcount из раздела 2.6.2. Она услужливо сообщает, что мы забыли написать документирующий комментарий для пакета:

```
$ go get github.com/golang/lint/golint
$ $GOPATH/bin/golint gopl.io/ch2/popcount
src/gopl.io/ch2/popcount/main.go:1:1:
    package comment should be of the form "Package popcount ..."
```

Команда go get поддерживает популярные сайты хостинга кода, такие как GitHub, Bitbucket и Launchpad, и может делать соответствующие запросы в их системы управления версиями. Для менее известных сайтов, возможно, в пути импорта придется указать, какой протокол контроля версий следует использовать, такой как Git или как Mercurial. Подробную информацию можно получить с помощью команды go help importpath.

Каталоги, которые создает `go get`, являются истинными клиентами удаленного хранилища, а не просто копиями файлов, поэтому вы можете использовать команды управления версиями для просмотра внесенных вами локальных изменений или выполнить обновление до другой версии. Например, каталог `golang.org/x/net` является клиентом Git:

```
$ cd $GOPATH/src/golang.org/x/net
$ git remote v
origin https://go.googlesource.com/net (fetch)
origin https://go.googlesource.com/net (push)
```

Обратите внимание на то, что имя домена в пути импорта пакета, `golang.org`, отличается от фактического имени домена сервера Git, `go.googlesource.com`. Это особенность инструмента `go`, которая позволяет пакетам применять пользовательское доменное имя в своих путях импорта при том, что размещены они в общей службе, такой как `googlesource.com` или `github.com`. HTML-страницы по адресу `https://golang.org/x/net/html` включают показанные ниже метаданные, которые перенаправляют инструмент `go` в Git-репозиторий на фактический сайт хостинга:

```
$ go build gopl.io/ch1/fetch
$ ./fetch https://golang.org/x/net/html | grep goimport
<meta name="goimport"
      content="golang.org/x/net git https://go.googlesource.com/net">
```

Если указан флаг `-u`, `go get` гарантирует, что все посещенные пакеты, включая зависимости, обновятся до последней версии, прежде чем будут выполнены сборка и установка. Без этого флага пакеты, уже имеющиеся локально, обновляться не будут.

Команда `go get -u`, как правило, извлекает последнюю версию каждого пакета, что очень удобно, когда вы начинаете работу, но может не подходить для развернутых проектов, в которых решающее значение имеет точный контроль зависимостей. Обычное решение этой проблемы заключается в *поставщике* кода, т.е. в том, чтобы сделать неизменяемую локальную копию всех необходимых зависимостей и обновлять ее очень осторожно и обдуманно. До Go 1.5 это требовало изменения путей импорта таких пакетов, так что наша копия `golang.org/x/net/html` превращалась в `gopl.io/vendor/golang.org/x/net/html`. Более поздние версии инструмента `go` поддерживают поставку непосредственно, хотя здесь у нас нет места, чтобы показать подробности. (См. *Vendor Directories* в выводе команды `go help gopath`.)

Упражнение 10.3. С помощью команды `fetch http://gopl.io/ch1/helloworld?go-get=1` выясните, какой сервис хранит образцы кода для данной книги. (HTTP-запросы от `go get` включают параметр `go-get`, так что серверы могут отличаться от указанных в обычном запросе браузера.)

10.7.3. Построение пакетов

Команда `go build` компилирует каждый пакет, указанный в качестве аргумента командной строки. Если пакет представляет собой библиотеку, результат игнорируется; проверяется только, чтобы при компиляции пакета не было ошибок компиляции.

Если пакет имеет имя `main`, `go build` вызывает компоновщик для создания выполнимого файла в текущем каталоге; имя выполнимого файла берется из последней части пути импорта пакета.

Поскольку каждый каталог содержит один пакет, каждая выполнимая программа (или *команда* в терминологии Unix) требует собственный каталог. Эти каталоги иногда являются дочерними по отношению к каталогу с именем `cmd`, как, например, `golang.org/x/tools/cmd/godoc` в случае команды, которая служит документацией пакетов Go через веб-интерфейс (раздел 10.7.4).

Пакеты могут быть указаны с помощью путей импорта, как мы видели выше, или с помощью относительного имени каталога, которое должно начинаться с `.` или `..`, даже если обычно это не требуется. Если никакие аргументы не предоставлены, предполагается текущий каталог. Таким образом, следующие команды строят один и тот же пакет, хотя каждая записывает выполнимый файл в каталог, в котором запускается `go build`:

```
$ cd $GOPATH/src/gopl.io/ch1/helloworld
$ go build
```

И

```
$ cd anywhere
$ go build gopl.io/ch1/helloworld
```

И

```
$ cd $GOPATH
$ go build ./src/gopl.io/ch1/helloworld
```

Но не

```
$ cd $GOPATH
$ go build src/gopl.io/ch1/helloworld
Ошибка: не найден пакет "src/gopl.io/ch1/helloworld".
```

Пакеты могут быть указаны и как список имен файлов, хотя это, как правило, используется только для небольших программ и разовых экспериментов. Если имя пакета — `main`, имя выполнимого файла берется из имени первого `.go`-файла:

```
$ cat quoteargs.go
package main

import (
    "fmt"
    "os"
)

func main() {
    fmt.Printf("%q\n", os.Args[1:])
}

$ go build quoteargs.go
```

```
$ ./quoteargs one "two three" four\ five
["one" "two three" "four five"]
```

Часто для разовых программ, таких как эта, мы хотим выполнить и сразу по построении запустить выполнимый файл. Команда `go run` сочетает в себе эти два шага:

```
$ go run quoteargs.go one "two three" four\ five
["one" "two three" "four five"]
```

Предполагается, что первый аргумент, который не оканчивается на `.go`, является началом списка аргументов для выполнимого файла.

По умолчанию команда `go build` строит запрошенный пакет и все его зависимости, а затем отбрасывает весь скомпилированный код за исключением окончательного выполнимого файла, если таковой имеется. Анализ зависимостей и компиляции выполняется на удивление быстро, но с ростом проектов до десятков пакетов и сотен тысяч строк кода время перекомпиляции зависимостей может стать заметным — потенциально достичь нескольких секунд, даже если эти зависимости не изменились.

Команда `go install` очень похожа на `go build`, с тем отличием, что она сохраняет скомпилированный код каждого пакета и команды вместо того, чтобы его отбросить. Скомпилированные пакеты сохраняются в подкаталогах каталога `$GOPATH/pkg`, соответствующего каталогу `src`, в котором хранятся исходные тексты, а выполнимые файлы хранятся в каталоге `$GOPATH/bin`. (Многие пользователи добавляют `$GOPATH/bin` в пути поиска выполнимых файлов.) После этого команды `go build` и `go install` не запускают компилятор для этих пакетов и команд, если они не изменились, что делает последующие построения гораздо более быстрыми. Для удобства `go build -i` устанавливает пакеты, от которых зависит целевой объект.

Поскольку скомпилированные пакеты зависят от платформы и архитектуры, `go install` сохраняет их в подкаталоге, имя которого включает значения переменных среды `GOOS` и `GOARCH`. Например, на компьютерах Mac пакет `golang.org/x/net/html` компилируется и устанавливается в файле `golang.org/x/net/html.a` в подкаталоге `$GOPATH/pkg/darwin_amd64`.

Очень просто выполняется в Go *кросс-компиляция*, т.е. построение выполнимого файла, предназначенного для работы с другой операционной системой или процессором. Просто установите переменные среды `GOOS` и `GOARCH` на время построения. Программа `cross` выводит операционную систему и архитектуру, для которой она была построена:

```
gopl.io/ch10/cross
func main() {
    fmt.Println(runtime.GOOS, runtime.GOARCH)
}
```

Следующие команды создают 64- и 32-разрядное приложения соответственно:

```
$ go build gopl.io/ch10/cross
$ ./cross
darwin amd64
```

```
$ GOARCH=386 go build gopl.io/ch10/cross
$ ./cross
darwin 386
```

Для некоторых пакетов может потребоваться компиляция различных версий кода для определенных платформ или процессоров, например для низкоуровневой переносимости или оптимизации версий важных процедур. Если имя файла содержит название операционной системы или имя архитектуры процессора наподобие `net_linux.go` или `asm_amd64.s`, то `go` будет компилировать файл только при построении для этой целевой системы или процессора. Специальные комментарии под названием *дескрипторы построения* обеспечивают более тонкое управление. Например, если файл содержит комментарий

```
// +build linux darwin
```

перед объявлением пакета (и его документирующим комментарием), `go build` будет компилировать его только при построении для Linux или Mac OS X, а следующий комментарий указывает, что данный файл никогда не должен компилироваться:

```
// +build ignore
```

Более подробную информацию можно найти в разделе *Build Constraints* документации пакета `go/build`:

```
$ go doc go/build
```

10.7.4. Документирование пакетов

Стиль Go настоятельно рекомендует тщательно документировать API пакетов. Непосредственно каждому объявлению экспортируемого члена пакета и самому объявлению пакета должен предшествовать комментарий, объясняющий его цель и использование.

Документирующие комментарии Go всегда являются полными предложениями, и первое предложение обычно представляет собой резюме, которое начинается с объявляемого имени. Параметры функции и другие идентификаторы упоминаются без кавычек. Например, вот как выглядит документирующий комментарий для `fmt.Fprintf`:

```
// Fprintf выполняет форматирование согласно спецификатору формата и
// запись в w. Возвращает количество записанных байтов и происшедшую
// ошибку записи.
func Fprintf(w io.Writer, format string, a ...interface{}) (int, error)
```

Детальная информация о форматировании функцией `Fprintf` разъясняется в документирующем комментарии самого пакета `fmt`. Комментарий, непосредственно предшествующий объявлению `package`, рассматривается как документирующий комментарий пакета в целом. Он должен быть только один, хотя может находиться в любом файле. Более длинный комментарий пакета может потребовать собственного

файла; комментарий пакета `fmt` состоит более чем из 300 строк. Этот файл обычно называется `doc.go`.

Хорошая документация не должна быть большой. Соглашения Go отдают предпочтение краткости и простоте в документации, как и во всем прочем, поскольку документация, как и код, также требует обслуживания. Многие объявления могут быть объяснены одним четко сформулированным предложением, а если поведение совершенно очевидно, комментарии не являются необходимыми.

Там, где в книге позволяло место, мы оставляли документирующие комментарии перед объявлениями, но вы можете найти куда лучшие примеры в стандартной библиотеке. Помочь вам сделать это могут два инструмента.

Инструмент `go doc` выводит объявление и документирующий комментарий для объекта, указанного в командной строке. Он может быть пакетом:

```
$ go doc time
package time // import "time"

Пакет time предоставляет функциональность
для измерения и вывода времени.

const Nanosecond Duration = 1 ...
func After(d Duration) <- chan Time
func Sleep(d Duration)
func Since(t Time) Duration
func Now() Time
type Duration int64
type Time struct { ... }
... и т.д. ...
```

Или членом пакета:

```
$ go doc time.Since
func Since(t Time) Duration
    Since возвращает время, прошедшее с момента t.
    Это аббревиатура для time.Now().Sub(t).
```

Или методом:

```
$ go doc time.Duration.Seconds
func (d Duration) Seconds() float64
    Seconds возвращает продолжительность в виде числа
    секунд с плавающей точкой.
```

Инструмент не требует завершения путей импорта или корректного регистра идентификаторов. Приведенная далее команда выводит документацию по `(*json.Decoder).Decode` из пакета `encoding/json`:

```
$ go doc json.decode
func (dec *Decoder) Decode(v interface{}) error
    Decode считывает очередное JSON-кодированное значение из входа
    и сохраняет его в значении, на которое указывает v.
```

Второй инструмент, не совсем верно названный `godoc`, предоставляет связанные HTML-страницы, которые содержат ту же информацию, что и `go doc`, а также многое другое. Сервер `godoc` по адресу `https://golang.org/pkg` охватывает стандартную библиотеку. На рис. 10.1 показана документация для пакета `time`, а в разделе 11.6 — интерактивный вывод примеров программ инструментом `godoc`. Сервер `godoc` по адресу `https://godoc.org` снабжен поиском в тысячах пакетов с открытым кодом.

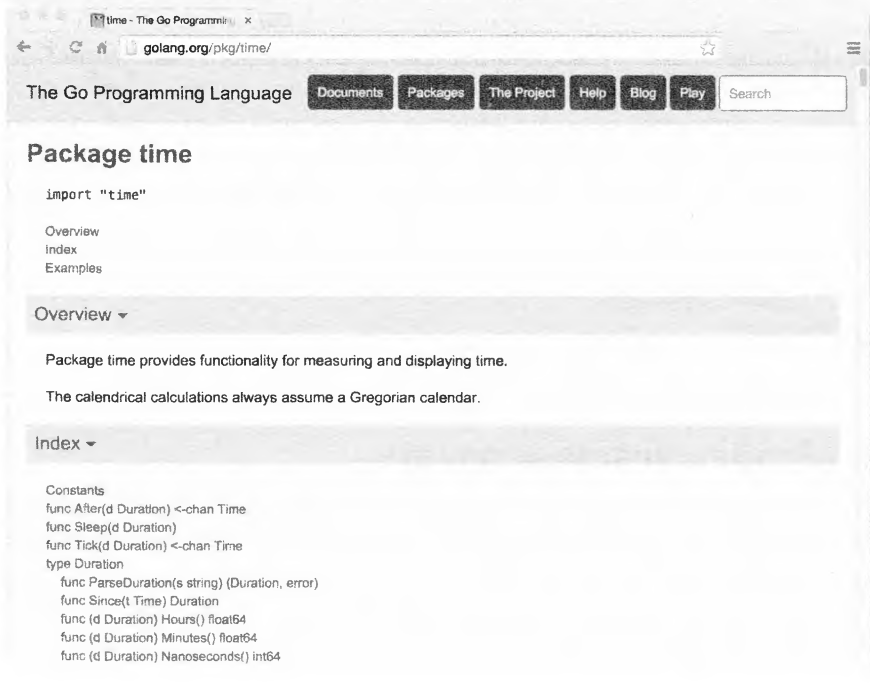


Рис. 10.1. Пакет `time` в `godoc`

Можно запустить экземпляр `godoc` в вашей рабочей области, чтобы просматривать собственные пакеты. После запуска приведенной ниже команды посетите адрес `http://localhost:8000/pkg` с помощью своего браузера.

```
$ godoc -http:8000
```

Флаги `-analysis=type` и `-analysis=pointer` дополняют документацию и исходный текст результатами "продвинутого" статистического анализа.

10.7.5. Внутренние пакеты

Пакет является наиболее важным механизмом инкапсуляции в программах Go. Неэкспортируемые идентификаторы видимы только в пределах одного пакета, а экспортируемые — видимы всем.

Впрочем, иногда полезна золотая середина — способ определения идентификаторов, которые являются видимыми для небольшого набора доверенных пакетов, но не

для всех. Например, разбивая большой пакет на более управляемые меньшие части, мы можем не захотеть раскрывать интерфейсы между этими частями для других пакетов. Или мы можем захотеть совместно использовать несколько вспомогательных функций несколькими пакетами проекта без их общедоступности. Или, возможно, мы просто хотим поэкспериментировать с новым пакетом без преждевременной фиксации его API, предоставив его "на испытательный срок" узкому кругу клиентов.

Для удовлетворения этих потребностей инструмент `go build` рассматривает пакет особым образом, если его путь импорта содержит часть с именем `internal`. Такие пакеты называются *внутренними пакетами*. Внутренний пакет может быть импортирован только другим пакетом, находящимся в дереве с корнем в родительском по отношению к `internal` каталоге. Например, для приведенных ниже пакетов `net/http/internal/chunked` может быть импортирован из `net/http/httputil` или `net/http`, но не из `net/url`. Однако `net/url` может импортировать `net/http/httputil`:

```
net/http
net/http/internal/chunked
net/http/httputil
net/url
```

10.7.6. Запрашиваемые пакеты

Инструмент `go list` выводит информацию о доступных пакетах. В своей простейшей форме `go list` проверяет, присутствует ли пакет в рабочем пространстве, и выводит его путь импорта, если это так:

```
$ go list github.com/go-sql-driver/mysql
github.com/go-sql-driver/mysql
```

Аргумент `go list` может содержать символы "`...`", которые соответствуют любой подстроке пути импорта пакета. Мы можем использовать их для перечисления всех пакетов в рабочей области Go:

```
$ go list ...
archive/tar
archive/zip
bufio
bytes
cmd/addr2line
cmd/api
... и т.д. ...
```

Или внутри конкретного поддерева:

```
$ go list gopl.io/ch3/...
gopl.io/ch3/basename1
gopl.io/ch3/basename2
gopl.io/ch3/comma
```

```
gopl.io/ch3/mandelbrot
gopl.io/ch3/netflag
gopl.io/ch3/printints
gopl.io/ch3/surface
```

Или связанных с конкретной темой:

```
$ go list ...xml...
encoding/xml
gopl.io/ch7/xmlselect
```

Команда `go list` получает полные метаданные для каждого пакета, а не только пути импорта, и делает эту информацию доступной для пользователей или других инструментов в различных форматах. Флаг `-json` заставляет `go list` вывести всю запись для каждого пакета в формате JSON:

```
$ go list json
hash
{
    "Dir": "/home/gopher/go/src/hash",
    "ImportPath": "hash",
    "Name": "hash",
    "Doc": "Package hash предоставляет интерфейсы для хеш-функций.",
    "Target": "/home/gopher/go/pkg/darwin_amd64/hash.a",
    "Goroot": true,
    "Standard": true,
    "Root": "/home/gopher/go",
    "GoFiles": [
        "hash.go"
    ],
    "Imports": [
        "io"
    ],
    "Deps": [
        "errors",
        "io",
        "runtime",
        "sync",
        "sync/atomic",
        "unsafe"
    ]
}
```

Флаг `-f` позволяет пользователям настраивать формат вывода, используя язык шаблонов пакета `text/template` (раздел 4.6). Следующая команда выводит транзитивные зависимости пакета `strconv`, разделенные пробелами:

```
$ go list -f '{{join .Deps " "}}' strconv
errors math runtime unicode/utf8 unsafe
```

А приведенная ниже команда выводит непосредственные импортирования каждого пакета в поддереве `compress` стандартной библиотеки:

```
$ go list -f '{{.ImportPath}} -> {{join .Imports " "}}' compress/...
compress/bzip2 -> bufio io sort
compress/flate -> bufio fmt io math sort strconv
compress/gzip -> bufio compress/flate errors fmt hash hash/crc32 io time
compress/lzw -> bufio errors fmt io
compress/zlib -> bufio compress/flate errors fmt hash hash/adler32 io
```

Команда `go list` полезна как для интерактивных запросов, так и для построения и тестирования сценариев автоматизации. Мы будем использовать ее снова в разделе 11.2.4. Для получения дополнительной информации, включая набор доступных полей и их смысл, обратитесь к результату выполнения команды `go help list`.

В этой главе рассмотрены все важные подкоманды инструмента `go`, за исключением одной. В следующей главе мы увидим, как команда `go test` используется для тестирования программ Go.

Упражнение 10.4. Создайте инструмент, который сообщает о множестве всех пакетов в рабочей области, которые транзитивно зависят от пакетов, указанных аргументами командной строки. *Указание*: вам нужно будет выполнить `go list` дважды: один раз — для исходных пакетов и один раз — для всех пакетов. Вы можете проанализировать вывод в формате JSON с помощью пакета `encoding/json` (раздел 4.5).

11

Тестирование

Морис Уилкс (Maurice Wilkes), разработчик EDSAC, первого компьютера с хранимой программой, в 1949 году столкнулся со случаем поразительного ясновидения, когда поднимался по лестнице в свою лабораторию. В своей книге *Memoirs of a Computer Pioneer* он пишет, как его "с необычайной силой охватила уверенность, что бо́льшая часть жизни будет потрачена на поиск ошибок в собственных программах". Наверняка каждый программист, работающий на компьютере с хранимой программой, отлично понимает Уилкса, хотя, возможно, не без некоторого удивления его наивности в вопросах построения сложного программного обеспечения.

Конечно, сегодняшние программы гораздо больше и сложнее, чем во времена Уилкса, и было затрачено множество усилий на разработку методов, которые сделали бы эту сложность сколь-нибудь управляемой. Своей эффективностью выделяются два метода. Во-первых, это рутинный коллегиальный просмотр программ, а во-вторых — предмет настоящей главы: тестирование.

Тестирование, под которым мы неявно подразумеваем *автоматизированное* тестирование, представляет собой практику написания небольших программ, которые проверяют, что тестируемый код (готовый код) ведет себя так, как от него и ожидается, для некоторых входных данных — либо тщательно отобранных для осуществления определенных возможностей, либо случайных для обеспечения наиболее широкого охвата всех вариантов.

Область тестирования программного обеспечения огромна. Задача тестирования отнимает у всех программистов некоторое их время, а у некоторых программистов — все их время. Литература по тестированию включает тысячи печатных книг и миллионы слов в блогах. Для каждого из основных языков программирования имеются десятки пакетов программного обеспечения, предназначенных для тестирования, некоторые — с привлечением серьезных теоретических разработок. Всего этого почти достаточно для того, чтобы убедить программистов, что для написания эффективных тестов они должны приобрести совершенно новый набор навыков.

По сравнению со всяческими современными технологиями подход Go к тестированию может показаться довольно устаревшим. Он опирается на единственную команду, `go test`, и ряд соглашений по написанию тестовых функций, которые запускает эта команда. Сравнительно легковесный механизм является легко расширяемым и достаточно эффективным для чистого тестирования.

На практике написание тестового кода не сильно отличается от написания самой оригинальной программы. Мы пишем короткие функции, которые сосредоточены на некоторой одной части задачи. Мы должны быть осторожны с граничными условиями, думать о структурах данных и понимать, какие результаты вычислений должны получаться из тех или иных входных данных. Но это тот же процесс, что и написание обычного кода Go; он не требует новых обозначений, соглашений и инструментария.

11.1. Инструмент go test

Подкоманда `go test` представляет собой тест-драйвер для пакетов Go, организованных в соответствии с некоторыми соглашениями. В каталоге пакета файлы, имена которых заканчиваются на `_test.go`, не являются частью пакета, который строится с помощью команды `go build`, но являются его частью при построении с помощью команды `go test`.

В файлах `*_test.go` три вида функций рассматриваются особым образом — это тесты, показатели производительности и примеры. *Тестовая функция*, которая представляет собой функцию с именем, начинающимся с `Test`, осуществляет некоторую программную логику для правильного поведения; `go test` вызывает тестовую функцию и сообщает о результате, который может быть либо `PASS` (пройден), либо `FAIL` (не пройден). *Функция производительности* имеет имя, начинающееся с `Benchmark`, и измеряет производительность некоторой операции; `go test` сообщает о среднем времени выполнения операции. Наконец *функция-пример*, имя которой начинается с `Example`, предоставляет машинно-проверяемую документацию. Мы будем подробно рассматривать тестовые функции в разделе 11.2, функции производительности — в разделе 11.4, а примеры — в разделе 11.6.

Команда `go test` сканирует файлы `*_test.go` на предмет наличия этих специальных функций, создает временный пакет `main`, который вызывает все их надлежащим образом, строит и запускает этот пакет, сообщает о полученных результатах и затем выполняет очистку.

11.2. Тестовые функции

Каждый тестовый файл должен импортировать пакет `testing`. Тестовые функции имеют следующую сигнатуру:

```
func TestName(t *testing.T) {
    // ...
}
```

Имена тестовых функций должны начинаться с `Test`; необязательный суффикс должен начинаться с прописной буквы:

```
func TestSin(t *testing.T) { /* ... */ }
func TestCos(t *testing.T) { /* ... */ }
func TestLog(t *testing.T) { /* ... */ }
```

Параметр t предоставляет методы для отчета о не пройденных тестах и для протоколирования дополнительной информации. Давайте определим пакет примера gopl.io/ch11/word1, содержащий одну функцию IsPalindrome, которая сообщает, является ли строка палиндромом, т.е. дает ли чтение ее в обратном направлении ту же строку. (Приведенная реализация проверяет каждый байт дважды, если строка является палиндромом; мы вернемся к этому вопросу в ближайшее время.)

gopl.io/ch11/word1

```
// Пакет word предоставляет утилиты для игр со словами.
package word

// IsPalindrome сообщает, является ли s палиндромом.
// (Первая попытка.)
func IsPalindrome(s string) bool {
    for i := range s {
        if s[i] != s[len(s)-1-i] {
            return false
        }
    }
    return true
}
```

Находящийся в том же каталоге файл word_test.go содержит две тестовые функции с именами TestPalindrome и TestNonPalindrome. Каждая из них проверяет, дает ли IsPalindrome правильный ответ для одного ввода, и сообщает о сбоях с помощью t.Error:

```
package word

import "testing"

func TestPalindrome(t *testing.T) {
    if !IsPalindrome("detartrated") {
        t.Error(`IsPalindrome("detartrated") = false`)
    }
    if !IsPalindrome("kayak") {
        t.Error(`IsPalindrome("kayak") = false`)
    }
}

func TestNonPalindrome(t *testing.T) {
    if IsPalindrome("palindrome") {
        t.Error(`IsPalindrome("palindrome") = true`)
    }
}
```

Команда go test (или go build) без аргумента, указывающего пакет, работает с пакетом в текущем каталоге. Мы можем строить и выполнять тесты с помощью следующей команды:

```
$ cd $GOPATH/src/gopl.io/ch11/word1
$ go test
ok   gopl.io/ch11/word1 0.008s
```

Довольные, мы отправляем программу заказчику, но не успеваем отметить это событие банкетом, как начинают прибывать сообщения об ошибках. Французский пользователь по имени Noelle Eve Elleon жалуется на то, что `IsPalindrome` не распознает слово "été". Другой, из Центральной Америки, разочарован тем, что программа отвергает фразу "A man, a plan, a canal: Panama". Эти конкретные сообщения естественным образом подходят для новых тестовых примеров:

```
func TestFrenchPalindrome(t *testing.T) {
    if !IsPalindrome("été") {
        t.Error(`IsPalindrome("été") = false`)
    }
}

func TestCanalPalindrome(t *testing.T) {
    input := "A man, a plan, a canal: Panama"
    if !IsPalindrome(input) {
        t.Errorf(`IsPalindrome(%q) = false`, input)
    }
}
```

Чтобы избежать написания длинной входной строки дважды, мы используем функцию `Errorf`, которая обеспечивает такое же форматирование, как и функция `Printf`.

После добавления двух новых тестов команда `go test` выдает информативные сообщения об ошибках:

```
$ go test
--- FAIL: TestFrenchPalindrome (0.00s)
    word_test.go:28: IsPalindrome("été") = false
--- FAIL: TestCanalPalindrome (0.00s)
    word_test.go:35: IsPalindrome("A man, a plan, a canal: Panama")
            = false
--- FAIL
FAIL gopl.io/ch11/word1 0.014s
```

Всегда следует сначала написать тест и убедиться, что он вызывает те же проблемы, что и в сообщении пользователя об ошибке. Только тогда мы можем быть уверены в том, что проблема действительно имеется и что она именно в нашей функции, и начать работать над ее исправлением.

Кроме того, запуск `go test` обычно выполняется быстрее, чем ручной проход по всем шагам, указанным в сообщении пользователя. Если набор тестов содержит много медленных проверок, мы можем ускорить тестирование, отбирая интересующие нас тестовые примеры.

Флаг `-v` приводит к выводу имени и времени выполнения каждого теста пакета:

```
$ go test -v
=== RUN TestPalindrome
--- PASS: TestPalindrome (0.00s)
=== RUN TestNonPalindrome
--- PASS: TestNonPalindrome (0.00s)
=== RUN TestFrenchPalindrome
--- FAIL: TestFrenchPalindrome (0.00s)
    word_test.go:28: IsPalindrome("été") = false
=== RUN TestCanalPalindrome
--- FAIL: TestCanalPalindrome (0.00s)
    word_test.go:35: IsPalindrome("A man, a plan, a canal: Panama")
            = false
FAIL
exit status 1
FAIL    gopl.io/ch11/word1 0.017s
```

А флаг `-run`, аргументом которого является регулярное выражение, приводит к тому, что команда `go test` выполняет только те тесты, имена функций которых соответствуют шаблону:

```
$ go test -v -run="French|Canal"
=== RUN TestFrenchPalindrome
--- FAIL: TestFrenchPalindrome (0.00s)
    word_test.go:28: IsPalindrome("été") = false
=== RUN TestCanalPalindrome
--- FAIL: TestCanalPalindrome (0.00s)
    word_test.go:35: IsPalindrome("A man, a plan, a canal: Panama")
            = false
FAIL
exit status 1
FAIL    gopl.io/ch11/word1 0.014s
```

Конечно, как только мы добьемся успешного прохождения выбранных тестов, мы должны пройти полный тест `go test` без флагов, чтобы убедиться, что, исправляя одну ошибку, мы не внесли другую.

Итак, мы убедились в наличии ошибок, и теперь наша задача состоит в том, чтобы их исправить. Быстрое исследование раскрывает причины первой ошибки: `IsPalindrome` использует последовательности байтов, а не рун, так что наличие не-ASCII символов (таких, как é в `"été"`) полностью дезориентирует эту функцию. Вторая ошибка возникает из-за того, что мы не игнорируем пробелы, знаки пунктуации и регистр букв.

Теперь мы перепишем функцию повнимательнее:

gopl.io/ch11/word2

```
// Пакет word предоставляет утилиты для игр со словами.
package word

import "unicode"
```

```
// IsPalindrome сообщает, является ли s палиндромом.
// Игнорируем регистр букв и символы, не являющиеся буквами.
func IsPalindrome(s string) bool {
    var letters []rune
    for _, r := range s {
        if unicode.IsLetter(r) {
            letters = append(letters, unicode.ToLower(r))
        }
    }
    for i := range letters {
        if letters[i] != letters[len(letters)-1-i] {
            return false
        }
    }
    return true
}
```

Мы также напишем более всеобъемлющий набор тестовых примеров, который собирает все предыдущие тестовые примеры и ряд новых в одну таблицу:

```
func TestIsPalindrome(t *testing.T) {
    var tests = []struct {
        input string
        want  bool
    }{
        {"", true},
        {"a", true},
        {"aa", true},
        {"ab", false},
        {"kayak", true},
        {"detartrated", true},
        {"A man, a plan, a canal: Panama", true},
        {"Evil I did dwell; lewd did I live.", true},
        {"Able was I ere I saw Elba", true},
        {"été", true},
        {"Et se resservir, ivresse reste.", true},
        {"palindrome", false},   // Не палиндром
        {"desserts", false},     // Полупалиндром
    }
    for _, test := range tests {
        if got := IsPalindrome(test.input); got != test.want {
            t.Errorf("IsPalindrome(%q) = %v", test.input, got)
        }
    }
}
```

Новый тест функция проходит успешно:

```
$ go test gopl.io/ch11/word2
ok gopl.io/ch11/word2 0.015s
```

Этот стиль *табличного тестирования* (table-driven testing) очень распространен в Go. Он обеспечивает простой способ добавления новых записей в таблицу при необходимости, а поскольку логика проверки и вывода сообщений не дублируется, мы можем приложить больше усилий для получения хорошего сообщения об ошибке.

Вывод при неудачном тесте *не* включает в себя весь стек в момент вызова `t.Errorf`. Функция `t.Errorf` не приводит к аварийной ситуации или остановке выполнения теста, в отличие от сбоев у многих средств тестирования для других языков. Тесты не зависят один от другого. Если ранние записи в таблице приводят к сбою теста, более поздние записи таблицы все равно будут проверены, так что мы можем узнать о нескольких сбоях при выполнении одного запуска теста.

Если нам действительно нужно остановить функцию тестирования, например, из-за сбоя некоторого кода инициализации или для предотвращения сообщения об ошибках, которые вызываются другими ошибками, мы используем функции `t.Fatal` и `t.Fatalf`. Они должны вызываться из той же go-подпрограммы, что и функция `Test`, но не из другой go-подпрограммы, созданной во время выполнения теста.

Сообщение об ошибке в тесте, как правило, имеет вид `"f(x) = y, требуется z"`, где `f(x)` объясняет, какая операция выполнялась и с какими входными данными, `y` — ее фактический результат, а `z` — ожидаемый. Там, где это удобно (как в нашем примере с палиндромом), в части `f(x)` используется синтаксис Go. Вывод `x` особенно важен в табличных тестах, так как данная проверка выполняется несколько раз с разными значениями. Избегайте избитых выражений и избыточной информации. Например, при проверке булевой функции, такой как `IsPalindrome`, часть "`требуется z`" можно опустить, поскольку она не добавляет никакой информации. Если `x`, `y` или `z` слишком длинное, выводите краткое резюме для соответствующих частей. Автору теста следует стремиться помочь программисту, которому придется диагностировать ошибки.

Упражнение 11.1. Напишите тесты для программы `charcount` из раздела 4.3.

Упражнение 11.2. Напишите набор тестов для множества `IntSet` (раздел 6.5), который проверяет, эквивалентно ли его поведение множеству на основе встроенных отображений. Сохраните свою реализацию для проверок производительности в упражнении 11.7.

11.2.1. Рандомизированное тестирование

Табличные тесты удобны для проверки, корректно ли функция работает с тщательно отобранными для проверки интересных случаев в логике функции входными данными. Другой подход, *рандомизированное тестирование*, исследует более широкий спектр входных данных путем их создания случайным образом.

Но как нам узнать, какой выход следует ожидать от нашей функции, учитывая случайность входа? Имеются две стратегии. Первая заключается в том, чтобы написать альтернативную реализацию функции, которая использует менее эффективный, но более простой и ясный алгоритм, и проверить, дают ли обе реализации один и тот же результат. Второй заключается в создании входных значений согласно некоторому шаблону, такому, для которого мы знаем, какой выход следует ожидать.

В приведенном ниже примере используется второй подход: функция `randomPalindrome` генерирует слова, которые являются палиндромами исходя из самого принципа их построения:

```
import "math/rand"

// randomPalindrome возвращает палиндром, длина и содержимое
// которого задаются генератором псевдослучайных чисел rng.
func randomPalindrome(rng *rand.Rand) string {
    n := rng.Intn(25) // Случайная длина до 24
    runes := make([]rune, n)
    for i := 0; i < (n+1)/2; i++ {
        r := rune(rng.Intn(0x1000)) // Случайная руна до '\u0999'
        runes[i] = r
        runes[n-1-i] = r
    }
    return string(runes)
}

func TestRandomPalindromes(t *testing.T) {
    // Инициализация генератора псевдослучайных чисел.
    seed := time.Now().UTC().UnixNano()
    t.Logf("ГПСЧ инициализирован: %d", seed)
    rng := rand.New(rand.NewSource(seed))
    for i := 0; i < 1000; i++ {
        p := randomPalindrome(rng)
        if !IsPalindrome(p) {
            t.Errorf("IsPalindrome(%q) = false", p)
        }
    }
}
```

Поскольку рандомизированные тесты являются недетерминированными, важно, чтобы при записи о сбоях в журнал выводилось достаточно информации, чтобы можно было воспроизвести этот сбой. В нашем примере входной параметр `p` для функции `IsPalindrome` дает нам всю необходимую информацию, но для функций, которые принимают более сложные входные данные, может быть проще записывать инициализирующее значение генератора псевдослучайных чисел (как сделано выше), чем дамп всех входных данных. Вооружившись этим инициализирующим значением, можно легко изменить тест так, чтобы детерминированно воспроизвести сбои.

Используя в качестве источника случайности текущее время, мы обеспечиваем тест новым набором входных данных при каждом очередном запуске в течение всего его жизненного цикла. Это особенно ценно, если ваш проект использует автоматизированную систему для периодического запуска всех тестов.

Упражнение 11.3. `TestRandomPalindromes` тестирует только палиндромы. Напишите рандомизированный тест для генерации и проверки *не* палиндромов.

Упражнение 11.4. Измените функцию `randomPalindrome` так, чтобы она проверяла, как функция `IsPalindrome` обрабатывает пробельные символы и знаки пунктуации.

11.2.2. Тестирование команд

Инструмент `go test` полезен для тестирования пакетов библиотек, но если приложить немного усилий, его можно использовать и для проверки команд. Пакет с именем `main` обычно создает выполнимую программу, однако он может также быть импортирован и как библиотека.

Давайте напишем тест для программы `echo` из раздела 2.3.2. Мы разделим программу на две функции: `echo` делает реальную работу, в то время как функция `main` выполняет чтение и анализ значений флагов и сообщает обо всех ошибках, возвращаемых функцией `echo`.

gopl.io/ch11/echo

```
// Echo выводит аргументы командной строки.
package main

import (
    "flag"
    "fmt"
    "io"
    "os"
    "strings"
)

var (
    n = flag.Bool("n", false, "опустить символы новой строки")
    s = flag.String("s", " ", "разделитель")
)
var out io.Writer = os.Stdout // Изменяется на время тестирования
func main() {
    flag.Parse()
    if err := echo(!*n, *s, flag.Args()); err != nil {
        fmt.Fprintf(os.Stderr, "echo: %v\n", err)
        os.Exit(1)
    }
}
func echo(newline bool, sep string, args []string) error {
    fmt.Fprint(out, strings.Join(args, sep))
    if newline {
        fmt.Fprintln(out)
    }
    return nil
}
```

Из тестовой функции мы будем вызывать `echo` с разнообразными аргументами и установками флагов и проверять, выдает ли функция в каждом конкретном случае верный вывод. Поэтому мы добавили в функцию `echo` параметры, позволяющие уменьшить зависимость от глобальных переменных. С учетом сказанного мы ввели также дополнительную глобальную переменную, `out`, типа `io.Writer`, в которую будет записываться результат. Если функция `echo` выполняет запись через эту переменную, а не непосредственно в `os.Stdout`, тесты могут подставить другую реализацию `Writer`, которая запоминает вывод для последующей проверки. Вот как выглядит тест в файле `echo_test.go`:

```
package main

import (
    "bytes"
    "fmt"
    "testing"
)

func TestEcho(t *testing.T) {
    var tests = []struct {
        newline bool
        sep     string
        args    []string
        want    string
    }{
        {true, "", []string{}, "\n"},
        {false, "", []string{}, ""},
        {true, "\t", []string{"one", "two", "three"},
               "one\ttwo\tthree\n"},
        {true, ",", []string{"a", "b", "c"}, "a,b,c\n"},
        {false, ":", []string{"1", "2", "3"}, "1:2:3"},
    }
    for _, test := range tests {
        descr := fmt.Sprintf("echo(%v, %q, %q)",
            test.newline, test.sep, test.args)
        out = new(bytes.Buffer) // Перехваченный вывод
        if err := echo(test.newline, test.sep, test.args); err != nil {
            t.Errorf("%s failed: %v", descr, err)
            continue
        }
        got := out.(*bytes.Buffer).String()
        if got != test.want {
            t.Errorf("%s = %q, требуется %q", descr, got, test.want)
        }
    }
}
```

Обратите внимание, что тестовый код находится в том же пакете, что и основной рабочий код. Хотя имя пакета — `main`, и в нем определена функция `main`, при тестировании этот пакет действует, как библиотека, которая предоставляет функцию `TestEcho` для тест-драйвера; его функция `main` игнорируется.

Организуя тест в виде таблицы, мы можем легко добавлять новые тестовые примеры. Давайте посмотрим, что произойдет, если тест не будет пройден, просто добавив в таблицу следующую строку:

```
// Примечание: неверно указанное ожидание!
{true, ",", []string{"a", "b", "c"}, "a b c\n"},
```

При этом `go test` выводит следующее:

```
$ go test gopl.io/ch11/echo
--- FAIL: TestEcho (0.00s)
    echo_test.go:31: echo(true, ",", ["a" "b" "c"]) = "a,b,c",
            требуется "a b c\n"
FAIL
FAIL    gopl.io/ch11/echo 0.006s
```

Сообщение об ошибке описывает неудачную операцию (с использованием Go-подобного синтаксиса), фактическое и ожидаемое поведения, в указанном порядке. Такое информативное сообщение об ошибке, как приведенное выше, обеспечивает довольно хорошее представление об основной причине проблемы даже до обращения к исходному коду теста.

Важно, что тестируемый код не вызывал `log.Fatal` или `os.Exit`, так как эти вызовы остановят работу тестирующего кода; вызов этих функций следует рассматривать как исключительное право функции `main`. Если произойдет что-то совершенно неожиданное и возникнет аварийная ситуация, тест-драйвер выполнит восстановление, хотя, конечно же, тест будет считаться не пройденным. Об ожидаемых ошибках, таких как получаемые в результате неверного пользовательского ввода, отсутствии файлов или неправильной конфигурации, следует сообщать, возвращая ненулевое значение ошибки. К счастью (хотя и к сожалению в качестве иллюстрации), наш пример `echo` настолько прост, что он никогда не будет возвращать ненулевую ошибку.

11.2.3. Тестирование белого ящика

Одним из способов классификации тестов является уровень требующихся им знаний о внутренней работе тестируемого пакета. Тестирование *черного ящика* предполагает, что о пакете неизвестно ничего, кроме предоставляемого им API и указанной в документации информации; внутренние компоненты пакета являются полностью непрозрачными. Напротив, тестирование *белого ящика* имеет привилегированный доступ к внутреннему устройству функций и структур данных пакета и может делать наблюдения и изменения, на которые не способен обычный клиент. Например, тест белого ящика может проверить, сохраняются ли инварианты типов данных пакета после каждой операции. (Название *белый ящик* используется по традиции, но куда точнее было бы название *прозрачный ящик*.)

Эти два подхода являются взаимодополняющими. Тесты черного ящика обычно более надежны и требуют меньшего количества обновлений по мере развития тестируемого программного обеспечения. Они также помогают автору теста проникнуться ощущениями клиента пакета, что облегчает выявление недостатков в дизайне API. Тестирование же белого ящика может обеспечить более детальный охват сложных частей реализации.

Мы уже видели примеры обоих типов. `TestIsPalindrome` вызывает только экспортируемую функцию `IsPalindrome` и, таким образом, является тестированием черного ящика. `TestEcho` вызывает функцию `echo` и обновляет глобальную переменную `out`, причем они обе не являются экспортируемыми, что делает данный тест тестом белого ящика.

При разработке `TestEcho` мы изменили функцию `echo` так, чтобы она использовала для вывода переменную уровня пакета `out` и тест мог заменить стандартный вывод альтернативной реализацией, которая записывает выводимые данные для последующего изучения. Используя ту же методику, мы можем заменить и другие части рабочего кода легко тестируемыми "поддельными" реализациями. Преимуществом таких поддельных реализаций является то, что они проще настраиваются, более предсказуемы, более надежны и проще в изучении. Они также могут избежать нежелательных побочных эффектов, таких как обновления рабочей базы данных или операций с кредитной картой.

В приведенном ниже коде показана логика проверки квот в веб-службе, которая обеспечивает сетевое хранилище пользовательских файлов. Когда пользователи превышают 90% от их квоты, система отправляет им предупреждение по электронной почте.

gopl.io/ch11/storage1

```
package storage

import (
    "fmt"
    "log"
    "net/smtp"
)

func bytesInUse(username string) int64 { return 0 /* ... */ }

// Настройка отправителя электронных писем.
// Примечание: никогда не помещайте пароль в исходный текст!
const sender   = "notifications@example.com"
const password = "correcthorsebatterystaple"
const hostname = "smtp.example.com"
const template = `Внимание, вы использовали %d байт хранилища,
                  %d%% вашей квоты.`

func CheckQuota(username string) {
    used := bytesInUse(username)
```

```
    const quota = 1000000000 // 1GB
    percent := 100 * used / quota
    if percent < 90 {
        return // OK
    }
    msg := fmt.Sprintf(template, used, percent)
    auth := smtp.PlainAuth("", sender, password, hostname)
    err := smtp.SendMail(hostname+":587", auth, sender,
                         []string{username}, []byte(msg))
    if err != nil {
        log.Printf("Сбой smtp.SendMail(%s): %s", username, err)
    }
}
```

Мы хотели бы протестировать этот код, но мы не хотим тестировать отправку электронной почты. Поэтому мы перемещаем логику отправления электронной почты в отдельную функцию и храним ее в неэкспортируемой переменной уровня пакета, `notifyUser`:

gopl.io/ch11/storage2

```
var notifyUser = func(username, msg string) {
    auth := smtp.PlainAuth("", sender, password, hostname)
    err := smtp.SendMail(hostname+":587", auth, sender,
        []string{username}, []byte(msg))
    if err != nil {
        log.Printf("Сбой smtp.SendEmail(%s): %s", username, err)
    }
}

func CheckQuota(username string) {
    used := bytesInUse(username)
    const quota = 1000000000 // 1GB
    percent := 100 * used / quota
    if percent < 90 {
        return // OK
    }
    msg := fmt.Sprintf(template, used, percent)
    notifyUser(username, msg)
}
```

Теперь мы можем написать тест, который заменяет отправление электронной почты функцией простого уведомления, которая просто регистрирует уведомляемого пользователя и содержимое сообщения:

```
package storage

import (
    "strings"
    "testing"
```

```
)

func TestCheckQuotaNotifiesUser(t *testing.T) {
    var notifiedUser, notifiedMsg string
    notifyUser = func(user, msg string) {
        notifiedUser, notifiedMsg = user, msg
    }
    // ... имитация условия 980-Мбайтной занятости ...
    const user = "joe@example.org"
    CheckQuota(user)
    if notifiedUser == "" && notifiedMsg == "" {
        t.Fatalf("notifyUser not called")
    }
    if notifiedUser != user {
        t.Errorf("Уведомлен (%s) вместо %s",
            notifiedUser, user)
    }
    const wantSubstring = "98% of your quota"
    if !strings.Contains(notifiedMsg, wantSubstring) {
        t.Errorf("неожиданное уведомление <<%s>>, "+
            "want substring %q", notifiedMsg, wantSubstring)
    }
}
```

Имеется одна проблема: после возврата из тестовой функции `CheckQuota` больше не работает так, как надо, потому что все еще использует поддельную тестовую реализацию `notifyUsers`. (При обновлении глобальных переменных всегда имеется риск такого рода.) Мы должны изменить тест, чтобы он восстанавливал предыдущее значение так, чтобы последующие тесты не наблюдали никакой замены, и должны сделать это на всех путях выполнения, включая сбои и аварийные ситуации в тестах. Это естественным образом приводит к применению `defer`.

```
func TestCheckQuotaNotifiesUser(t *testing.T) {
    // Сохранение и восстановление исходного значения notifyUser.
    saved := notifyUser
    defer func() { notifyUser = saved }()

    // Установка поддельной функции для notifyUser.
    var notifiedUser, notifiedMsg string
    notifyUser = func(user, msg string) {
        notifiedUser, notifiedMsg = user, msg
    }
    // ... остальная часть теста ...
}
```

Этот шаблон может использоваться для временного сохранения и восстановления глобальных переменных всех видов, включая флаги командной строки, параметры отладки и параметры производительности; для установки и удаления перехватчиков,

которые заставляют рабочий код вызывать некоторый тестовый код, когда происходит что-то интересное; а также при таких редких, но важных ситуациях, как тайм-ауты, ошибки и даже конкретные чередования параллельных действий.

Использование глобальных переменных таким образом является безопасным только потому, что `go test` обычно не запускает несколько тестов одновременно.

11.2.4. Внешние тестовые пакеты

Рассмотрим пакеты `net/url`, которые предоставляет анализатор URL, и `net/http`, который обеспечивает веб-сервер и клиентскую библиотеку HTTP. Как мы могли бы ожидать, более высокоуровневый пакет `net/http` зависит от пакета более низкого уровня `net/url`. Однако один из тестов в `net/url` представляет собой пример, демонстрирующий взаимодействие между URL и клиентской библиотекой HTTP. Другими словами, тест пакета нижнего уровня импортирует пакет более высокого уровня.

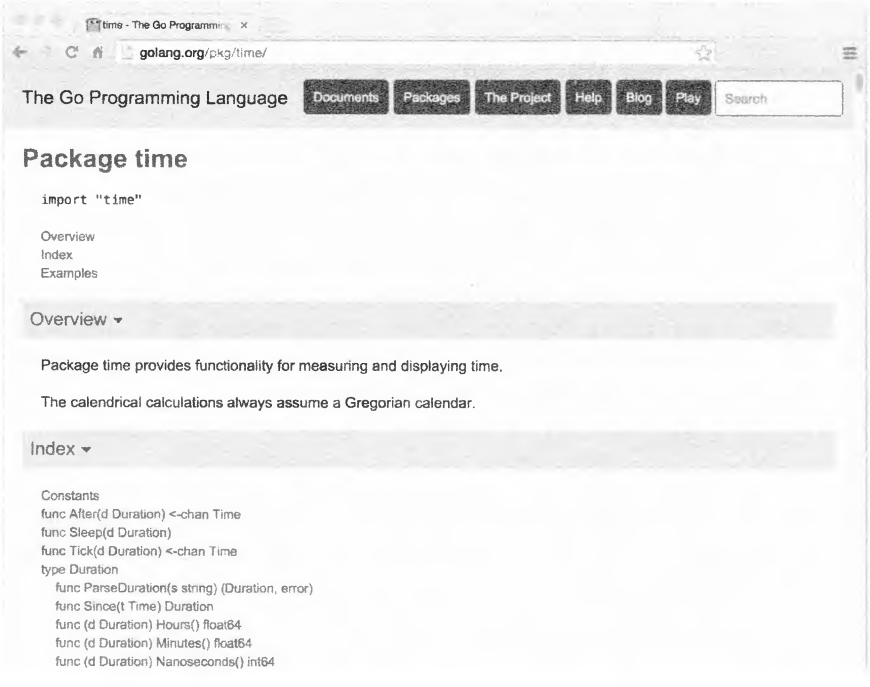


Рис. 11.1. Тест `net/url` зависит от теста `net/http`

Объявление этой тестовой функции в пакете `net/url` будет создавать цикл в графе импорта пакетов, как показано стрелкой, ведущей вверх на рис. 11.1, но, как мы говорили в разделе 10.1, спецификация Go запрещает циклический импорт.

Мы решаем эту проблему, объявляя тестирующую функцию во *внешнем тестовом пакете*, т.е. в файле в каталоге `net/url`, объявление пакета которого выглядит как `package url_test`. Дополнительный суффикс `_test` является сигналом для

go test о том, что он должен создать дополнительный пакет, содержащий только эти файлы, и выполнять его тесты. Можно рассматривать этот внешний тестовый пакет как пакет, который имеет путь импорта net/url_test, но не может быть импортирован с этим или любым иным именем.

Поскольку внешние тесты находятся в отдельном пакете, они могут импортировать вспомогательные пакеты, которые также зависят от пакета, который они тестируют; обычный тестовый пакет на такое не способен. С точки зрения проектирования внешний тестовый пакет логически находится выше, чем оба пакета, от которых он зависит, как показано на рис. 11.2.

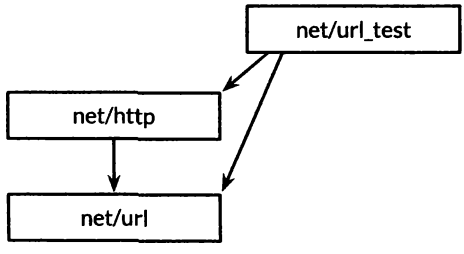


Рис. 11.2. Внешний тестовый пакет разрушает цикл зависимостей

Избегая циклов импорта, внешние тестовые пакеты позволяют тестам, в особенности *тестам интеграции* (которые тестируют взаимодействие нескольких компонентов), свободно импортировать другие пакеты так, как это делает приложение.

Мы можем использовать инструмент go list, чтобы выяснить, какие исходные файлы Go в каталоге пакета являются рабочим кодом, какие — тестами в текущем пакете, а какие — внешними тестами. В качестве примера воспользуемся пакетом fmt. GoFiles представляет собой список файлов, которые содержат рабочий код; это файлы, которые go build будет включать в приложение:

```
$ go list -f={{.GoFiles}} fmt
[doc.go format.go print.go scan.go]
```

TestGoFiles представляет собой список файлов, которые также принадлежат пакету fmt, но эти файлы, имена которых оканчиваются как _test.go, включаются только при построении тестов:

```
$ go list -f={{.TestGoFiles}} fmt
[export_test.go]
```

В таких файлах обычно располагаются тесты пакета, так что их отсутствие в пакете fmt необычно; назначение файла export_test.go мы вскоре поясним.

XTestGoFiles представляет собой список файлов, которые составляют пакет внешнего тестирования, fmt_test, так что эти файлы должны импортировать пакет fmt для того, чтобы его использовать. Они также включаются в построение только в процессе тестирования:

```
$ go list -f={{.XTestGoFiles}} fmt
[fmt_test.go scan_test.go stringer_test.go]
```

Иногда пакету внешнего тестирования может потребоваться привилегированный доступ к внутреннему представлению тестируемого пакета, если, например, тест белого ящика должен находиться в отдельном пакете во избежание цикла импорта. В таких случаях мы используем следующий трюк: добавляем объявления в файл `_test.go` в тестируемом пакете для предоставления необходимого внутреннего представления внешнему тесту. Таким образом, этот файл, по сути, представляет собой "черный ход" в пакет. Если исходный файл существует только для этой цели и не содержит тестов, его часто называют `export_test.go`.

Например, реализации пакета `fmt` требуется функциональность `unicode.IsSpace` — как часть `fmt.Scanf`. Чтобы избежать создания нежелательных зависимостей, `fmt` не импортирует пакет `unicode` и его большие таблицы данных; вместо этого он содержит более простую реализацию, которая называется `isSpace`.

Чтобы обеспечить согласованность поведения `fmt.isSpace` и `unicode.IsSpace`, `fmt` предусмотрительно содержит тест. Это внешнее тестирование, и, таким образом, оно не может получить доступ к `isSpace` непосредственно. Поэтому `fmt` открывает черный ход к этой функции путем объявления экспортируемой переменной, которая содержит внутреннюю функцию `isSpace`. В этом и состоит весь файл `export_test.go` пакета `fmt`.

```
package fmt
var IsSpace = isSpace
```

Этот тестовый файл не определяет никаких тестов; он просто объявляет экспортируемый символ `fmt.IsSpace` для использования внешним тестированием. Этот трюк может также использоваться всякий раз, когда внешнему тестированию необходимо использовать некоторые из методов тестирования белого ящика.

11.2.5. Написание эффективных тестов

Многие новички удивляются минимализму каркаса тестирования Go. Каркасы других языков обеспечивают механизмы для идентификации тестовых функций (часто с помощью рефлексии или метаданных), перехваты для выполнения различных операций до и после запуска тестов, библиотеки вспомогательных функций для проверки распространенных предикатов, сравнения значений, форматирование сообщений об ошибках и прерывания непройденных тестов (зачастую с помощью исключения). Хотя эти механизмы могут сделать тесты очень краткими, в результате создается впечатление, что эти тесты написаны на совершенно ином языке. Кроме того, хотя они могут корректно сообщать о том, пройден ли тест, их сообщения могут быть весьма загадочными, с сообщениями вида `"assert: 0 == 1"` или многостраничными трассировками стека.

Отношение Go к тестированию представляет разительный контраст. Он ожидает от авторов тестов, что они сами сделают бо́льшую часть работы, так же, как и при разработке обычных программ. Процесс тестирования не является сугубо механиче-

ским действием; тест также имеет пользовательский интерфейс, хотя и предназначенный только для тех, кто занимается сопровождением программы. Хороший тест выводит четкое и сжатое описание симптомов проблемы и, возможно некоторую другую информацию о контексте, в котором произошел сбой. В идеале программисту не требуется читать исходный текст, чтобы расшифровать сообщения теста. Хороший тест не должны завершаться после первого же сбоя; он должен попытаться представить за один проход несколько ошибок, поскольку общая картина сбоев может помочь выявлению проблемы.

Приведенная ниже функция сравнивает два значения, создает обобщенное сообщение об ошибке и останавливает программу. Это просто в использовании и корректно, но сообщение об ошибке при сбое практически бесполезно. Данная функция не решает сложную проблему предоставления хорошего пользовательского интерфейса:

```
import (
    "fmt"
    "strings"
    "testing"
)

// Плохая функция.
func assertEqual(x, y int) {
    if x != y {
        panic(fmt.Sprintf("%d != %d", x, y))
    }
}

func TestSplit(t *testing.T) {
    words := strings.Split("a:b:c", ":")
    assertEqual(len(words), 3)
    // ...
}
```

В определенном смысле приведенная функция страдает от *преждевременной абстракции*: рассматривая отказ этого конкретного теста как простое различие двух целых чисел, мы теряем возможность получения значащего контекста. Можно предоставить сообщение и получше, с конкретной информацией, как показано в примере ниже. Время для ввода абстракций наступает только после того, как в данном тесте наблюдается один и тот же повторяющийся шаблон ошибки:

```
func TestSplit(t *testing.T) {
    s, sep := "a:b:c", ":"
    words := strings.Split(s, sep)
    if got, want := len(words), 3; got != want {
        t.Errorf("Split(%q, %q) возвращает %d слов, а требуется %d",
                 s, sep, got, want)
    }
    // ...
}
```

Теперь тест сообщает о том, какая функция вызывалась, ее входные данные и результаты — тест явно указывает фактическое и ожидаемое значения результата. Тест продолжает выполняться, даже если проверка указывает на наличие сбоя. После того как мы написали тест наподобие рассматриваемого, естественный следующий шаг зачастую состоит не в том, чтобы определить функцию, заменяющую всю конструкцию `if`, а в том, чтобы выполнить тест в цикле, в котором варьируются значения `s`, `sep` и `want`, как в табличном тесте функции `IsPalindrome`.

Предыдущий пример не нуждается в каких-либо служебных функциях, но, конечно, это не должно удерживать нас от добавления функций, если они помогают упростить код. (Мы рассмотрим одну такую вспомогательную функцию, `reflect.DeepEqual`, в разделе 13.3.) Ключ к хорошему тесту — начать с реализации конкретного необходимого вам поведения и лишь затем использовать функции для упрощения кода и устранения повторений. Если начинать с библиотеки абстрактных, обобщенных тестовых функций, наилучшие результаты получаются редко.

Упражнение 11.5. Расширьте `TestSplit` так, чтобы она использовала таблицу входных и ожидаемых выходных данных.

11.2.6. Избегайте хрупких тестов

Приложение, которое часто сбоит, встречаясь с новыми, хотя и допустимыми входными данными, называется *глючным* (buggy); тест, который дает ложные сбои при внесении корректных изменений в программу, называется *хрупким* (brittle). Так же, как глючная программа расстраивает своих пользователей, хрупкий тест раздражает программиста, сопровождающего код. Наиболее хрупкие тесты, которые сбоят при почти любом изменении (верном или неверном) рабочего кода, иногда называют *детекторами изменений* или *тестами статус-кво*, и время, которое приходится затрачивать на работу с ними, очень быстро сводит на нет всю выгоду, которую они, казалось бы, должны предоставлять.

Когда тестируемая функция дает сложный вывод, такой как длинная строка, сложная структура данных или файл, так и хочется убедиться, что результат в точности равен некоторому "золотому" значению, которое ожидалось во время написания теста. Но по мере развития программы части выходных данных, скорее всего, будут изменяться. И это касается не только выходных данных; функции со сложными входными данными часто дают сбои из-за того, что используемые в тесте входные данные больше не являются корректными.

Самый простой способ избежать хрупких тестов — проверять только те свойства, которые вас интересуют. Тестируйте более простые и стабильные интерфейсы программы вместо внутренних функций. Будьте избирательны в ваших проверках. Например, не стоит проверять точное совпадение строк, но следует следить за важными подстроками, которые остаются неизменными по мере развития программы. Часто имеет смысл написать большую функцию для того, чтобы извлечь из сложного вывода саму его суть, чтобы получить возможность надежной проверки. Хотя эта работа может показаться требующей больших усилий, она может быстро окупиться за счет

экономии времени, которое в противном случае было бы затрачено на работу с ложно срабатывающими тестами.

11.3. Охват

По своей природе тестирование никогда не является полным. Как выразился известный ученый-кибернетик Эдсгер Дейкстра (Edsger Dijkstra), "тестирование показывает присутствие, не отсутствие ошибок". Никакое количество тестов не может доказать отсутствие ошибок в пакете. В лучшем случае они увеличивают нашу уверенность, что этот пакет хорошо работает в широком спектре важных сценариев.

Степень, до которой набор тестов исследует тестируемый пакет, называется *охватом* теста. Охват нельзя непосредственно выразить количественно (динамика любых сколь-нибудь нетривиальных программ выходит за рамки точного измерения), но имеются эвристики, которые могут помочь нам направить наши усилия по тестированию в том направлении, в котором они окажутся наиболее полезными.

Самой простой и наиболее широко используемой среди этих эвристик является эвристика *охвата инструкций*. Охват инструкций тестом представляет собой долю инструкций в исходном тексте, которые во время теста выполняются по крайней мере один раз. В этом разделе мы будем использовать инструмент Go `cover` (который интегрирован в `go test`), чтобы измерить охват инструкций и выявить очевидные пробелы в тестах.Приведенный ниже код является табличным тестом для вычислителя выражений, который мы строили в главе 7, "Интерфейсы":

gopl.io/ch7/eval

```
func TestCoverage(t *testing.T) {
    var tests = []struct {
        input string
        env   Env
        want  string // Ожидаемая ошибка от Parse/Check
                     // или результат Eval
    }{
        {"x % 2", nil, "неожиданный символ '%'"},
        {"!true", nil, "неожиданный символ '!'"},
        {"log(10)", nil, `неизвестная функция "log"`},
        {"sqrt(1, 2)", nil, "вызов sqrt с 2 аргументами, нужен 1"},
        {"sqrt(A / pi)", Env{"A": 87616, "pi": math.Pi}, "167"},
        {"pow(x, 3) + pow(y, 3)", Env{"x": 9, "y": 10}, "1729"},
        {"5 / 9 * (F - 32)", Env{"F": -40}, "-40"},
    }
    for _, test := range tests {
        expr, err := Parse(test.input)
        if err == nil {
            err = expr.Check(map[Var]bool{})
        }
        if err != nil {
```

```
            if err.Error() != test.want {
                t.Errorf("%s: получено %q, требуется %q",
                         test.input, err, test.want)
            }
            continue
        }
        got := fmt.Sprintf("%.6g", expr.Eval(test.env))
        if got != test.want {
            t.Errorf("%s: %v => %s, want %s",
                test.input, test.env, got, test.want)
        }
    }
}
```

Сначала проверим, что тест пройден:

```
$ go test -v -run=Coverage gopl.io/ch7/eval
=== RUN TestCoverage
--- PASS: TestCoverage (0.00s)
PASS
ok    gopl.io/ch7/eval 0.011s
```

Следующая команда выводит сообщение об использовании инструмента охвата (в переводе):

```
$ go tool cover
Использование 'go tool cover':
Профиль охвата, генерируемый 'go test':
    go test -coverprofile=c.out

Открыть браузер, выводящий аннотированный исходный текст:
    go tool cover -html=c.out
...
```

Команда `go tool` запускает один из выполнимых файлов инструментария Go. Эти программы находятся в каталоге `$GOROOT/pkg/tool/${GOOS}_${GOARCH}`. Благодаря `go build` нам редко приходится обращаться к ним непосредственно.

Теперь запустим тест с флагом `-coverprofile`:

```
$ go test -run=Coverage -coverprofile=c.out gopl.io/ch7/eval
ok    gopl.io/ch7/eval 0.032s охват: 68.5% инструкций
```

Этот флаг включает сбор данных об охвате посредством *измерений* (instrumenting) производственного кода, т.е. он изменяет копию исходного кода так, чтобы перед выполнением каждого блока инструкций устанавливалась некоторая булева переменная, по одной переменной на каждый блок. Непосредственно перед выходом из измененной таким образом программы значения всех переменных записываются в указанный журнальный файл `c.out` и выводится итоговое значение доли инструкций, которые были выполнены. (Если все, что вам нужно, — это итоговые значения, воспользуйтесь командой `go test cover`.)

Если запустить `go test` с флагом `-covermode=count`, при выполнении каждого блока вместо установки булева значения будет увеличиваться соответствующий счетчик. В журнале будут указаны количественные величины для каждого блока, которые позволят определить, какие блоки выполняются более часто, а какие — реже.

Собрав данные, мы можем запустить инструмент `cover`, который обрабатывает журнал, создает HTML-отчет и открывает его в новом окне браузера (рис. 11.3):

```
$ go tool cover -html=c.out
```

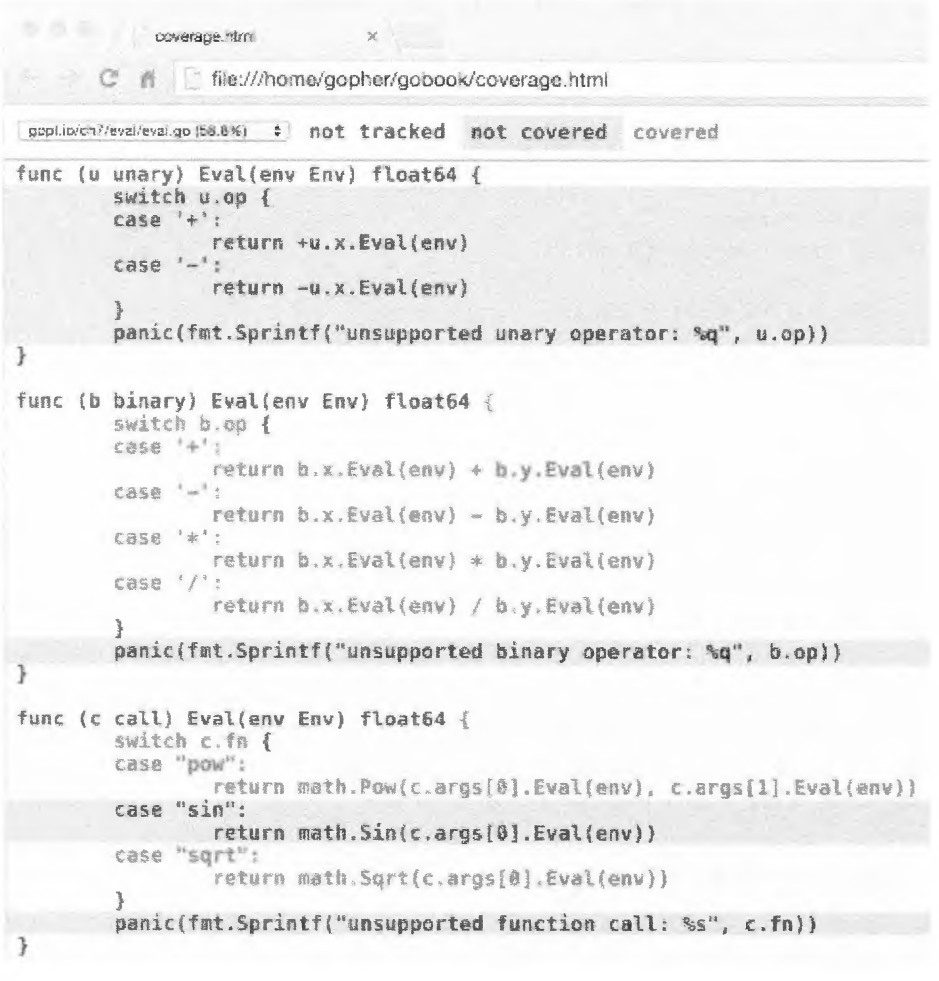


Рис. 11.3. Отчет об охвате

Каждая инструкция окрашена либо в зеленый цвет, если она была охвачена, либо в красный, если не была (для ясности на рисунке красный текст приведен на затененном фоне). Мы сразу же видим, что ни одни из наших входных данных не приводят к вызову метода `Eval` унарного оператора. Если мы добавим в таблицу новый тесто-

вый пример и вновь выполним две предыдущие команды, код унарного выражения станет зеленым:

```
{"-x * -x", eval.Env{"x": 2}, "4"}
```

Однако две инструкции `panic` остаются красными. Это не должно быть удивительным, потому что предполагается, что данные инструкции недоступны.

Достижение 100% охвата инструкций звучит как благородная цель, но обычно она не осуществима на практике и не представляется разумной тратой сил. То, что инструкция выполняется, не означает, что в ней нет ошибок; чтобы охватить все интересные случаи, сложные выражения должны выполняться много раз с различными входными данными. Одни инструкции, такие как показанные выше инструкции `panic`, никогда не достигаются. Другие, такие как обрабатывающие редкие ошибки, трудно выполнить, но они редко достижимы на практике. Тестирование является принципиально прагматичной деятельностью, компромиссом между стоимостью написания тестов и стоимостью сбоев, которые можно предотвратить путем тестирования. Инструменты, связанные с охватом, могут помочь определить слабые места, но разработка хороших тестовых примеров требует таких же тщательных размышлений, как и программирование в целом.

11.4. Функции производительности

Эталонное тестирования представляет собой практику измерения производительности программы для фиксированной рабочей нагрузки. В Go функция производительности выглядит как тестовая функция, но с префиксом `Benchmark` и параметром `*testing.B`, который предоставляет большую часть тех же методов, что и `*testing.T`, плюс несколько дополнительных, связанных с измерением производительности. Он также предоставляет целочисленное поле `N`, которое указывает количество раз измеряемой операции.

Вот как выглядит функция тестирования для `IsPalindrome`, вызывающая эту функцию в цикле `N` раз:

```
import "testing"
func BenchmarkIsPalindrome(b *testing.B) {
    for i := 0; i < b.N; i++ {
        IsPalindrome("A man, a plan, a canal: Panama")
    }
}
```

Мы запускаем ее с помощью приведенной ниже команды. В отличие от тестов, функции производительности по умолчанию не запускаются. Аргумент флага `-bench` выбирает, какие функции производительности будут запущены. Это регулярное выражение, соответствующее именам функций `Benchmark`, со значением по умолчанию, которое не соответствует ни одной из них. Шаблон "`.`" приводит к соответствию всем функциям в пакете `word`, но так как есть только одна функция, эквивалентная запись имеет вид `-bench=IsPalindrome`:

```
$ cd $GOPATH/src/gopl.io/ch11/word2
$ go test -bench=.
PASS
BenchmarkIsPalindrome-8  1000000     1035 ns/op
ok    gopl.io/ch11/word2        2.179s
```

Числовой суффикс имени функции производительности (здесь — 8) указывает значение GOMAXPROCS, которое является важным для параллельных тестов производительности.

Отчет говорит, что каждый вызов `IsPalindrome` занимает около 1.035 микросекунды при усреднении по 1 000 000 запусков. Поскольку изначально функции производительности не представляют, сколько времени выполняется операция, она делает первоначальные измерения для небольшого значения N, а затем экстраполирует их для получения значения, достаточно большого для выполнения устойчивого измерения времени.

Причина, по которой цикл реализуется функцией производительности, а не вызывающим кодом в драйвере теста, заключается в том, что функция производительности имеет возможность выполнения любого необходимого одноразового кода настройки вне цикла, без добавления его к измеряемому времени на каждой итерации. Если этот код настройки нарушает результаты тестирования, параметр `testing.B` предоставляет методы для остановки, возобновления и сброса таймера, но они требуются очень редко.

Теперь, когда у нас есть результаты эталонных тестов и тестов, легко попробовать идеи для того, чтобы сделать программу быстрее. Возможно, наиболее очевидная оптимизация — останавливать второй цикл `IsPalindrome` посредине, чтобы избежать повторных сравнений:

```
n := len(letters)/2
for i := 0; i < n; i++ {
    if letters[i] != letters[len(letters)-1-i]
    {
        return false
    }
}
return true
```

Но как это часто бывает, очевидная оптимизация не всегда дает ожидаемые выгоды. Данная оптимизация дала лишь 4% улучшения в одном эксперименте:

```
$ go test -bench=.
PASS
BenchmarkIsPalindrome-8    1000000    992 ns/op
ok    gopl.io/ch11/word2    2.093s
```

Другая идея заключается в том, чтобы предварительно выделить достаточно большой массив для `letters`, вместо того чтобы расширять его последовательными вызовами append. Объявление `letters` как массива подходящего размера, как здесь,

```
letters := make([]rune, 0, len(s))
for _, r := range s {
    if unicode.IsLetter(r) {
        letters = append(letters, unicode.ToLower(r))
    }
}
```

дает улучшение почти на 35%, и теперь показатели производительности усредняются по 2 000 000 итераций:

```
$ go test -bench=.
PASS
BenchmarkIsPalindrome-8    2000000    697 ns/op
ok    gopl.io/ch11/word2    1.468s
```

Как показывает данный пример, наиболее быстрой программой часто оказывается та, которая делает наименьшее количество выделений памяти. Флаг командной строки `-benchmem` включает в отчет статистику распределений памяти. Давайте сравним количество выделений до оптимизации:

```
$ go test -bench=. -benchmem
PASS
BenchmarkIsPalindrome   1000000   1026 ns/op   304 B/op   4 allocs/op
```

и после нее:

```
$ go test -bench=. -benchmem
PASS
BenchmarkIsPalindrome   2000000   807 ns/op   128 B/op   1 allocs/op
```

Сборка всех распределений памяти в один вызов `make` ликвидировала 75% распределений и вдвое уменьшила количество расходуемой памяти.

Функции производительности говорят нам об абсолютном времени, затрачиваемом на некоторую операцию, но во многих случаях нас интересует *относительное* время выполнения двух различных операций. Например, если функция тратит 1 мс для обработки 1000 элементов, то сколько времени займет обработка 10 000 или миллиона? Такие сравнения показывают асимптотический рост времени работы функции. Другой пример: каков наилучший размер буфера ввода-вывода? Измерение производительности для разных размеров может помочь нам выбрать наименьший буфер, который обеспечивает удовлетворительные результаты. Третий пример: какой алгоритм наилучшим образом подходит для выполнения данной работы? Изучение двух различных алгоритмов для одних и тех же входных данных зачастую может показать сильные и слабые стороны каждого из них при важных или представительных входных данных.

Сравнительные показатели достигаются с помощью обычного кода. Они обычно принимают вид одной параметризованной функции, вызываемой из нескольких функций `Benchmark` с различными значениями, например:

```
func benchmark(b *testing.B, size int) { /* ... */ }
func Benchmark10(b *testing.B) { benchmark(b, 10) }
```

```
func Benchmark100(b *testing.B) { benchmark(b, 100) }
func Benchmark1000(b *testing.B) { benchmark(b, 1000) }
```

Параметр `size`, который определяет размер входных данных, варьируется для разных функций, но является константой в пределах каждой функции производительности. Постарайтесь противостоять искушению использовать параметр `b.N` как размер входных данных. Если только вы не интерпретируете его как количество итераций для входных данных фиксированного размера, результаты вашего исследования окажутся бессмысленными.

Шаблоны, выявляемые при сравнении результатов функций производительности, особенно полезны во время разработки программы, но их не следует игнорировать и при рабочей программе. По мере развития программы могут расти ее входные данные, она может устанавливаться на новых операционных системах или процессорах с различными характеристиками, и мы можем повторно использовать эти данные для пересмотра проектных решений.

Упражнение 11.6. Напишите функцию производительности для сравнения реализации `PopCount` из раздела 2.6.2 с вашими решениями упражнений 2.4 и 2.5. В какой момент не срабатывает даже табличное тестирование?

Упражнение 11.7. Напишите функции производительности для `Add`, `UnionWith` и других методов `*IntSet` (раздел 6.5) с использованием больших псевдослучайных входных данных. Насколько быстрыми вы сможете сделать эти методы? Как влияет на производительность выбор размера слова? Насколько быстро работает `IntSet` по сравнению с реализацией множества на основе отображения?

11.5. Профилирование

Контрольные показатели полезны для измерения производительности конкретных операций, но когда мы пытаемся сделать медленную программу более быстрой, у нас зачастую нет никаких идей, с чего начать. Каждый программист знает афоризм Дональда Кнута (Donald Knuth) о преждевременной оптимизации, который появился в его работе "*Структурное программирования с инструкцией goto*" в 1974 году. Хотя его часто неверно воспринимают как указание, что производительность не имеет значения, в его исходном контексте мы можем разглядеть другой смысл.

> Не существует никаких сомнений в том, что Грааль эффективности приводит к злоупотреблениям. Программисты тратят огромное количество времени, размышляя или беспокоясь о скорости некритических частей их программ, и эти попытки достичь эффективности на самом деле оказывают сильное отрицательное воздействие на отладку и обслуживание. Мы *должны* забыть о малой эффективности, скажем, примерно на 97% времени: преждевременная оптимизация является корнем всех зол.
>
> Однако мы не должны пренебрегать нашими возможностями в рамках критических 3%. Хороший программист не будет самоуспокоен такими рассуждениями. Он будет достаточно мудрым, чтобы внимательно рассмотреть критический код,

но только *после* того, как этот код будет обнаружен. Серьезная ошибка программистов заключается в априорных суждениях о том, какие части программы действительно критические, поскольку опыт применения инструментов измерения производительности слишком часто противоречит интуитивным догадкам.

Если мы хотим внимательно рассмотреть скорость своих программ, наилучший метод для определения критического кода — *профилирование*. Профилирование представляет собой автоматизированный подход к измерению производительности на основе выборки ряда *событий* во время выполнения с последующей их экстраполяцией во время последующей обработки. Получаемая в результате статистическая сводка называется *профилем*.

Go поддерживает многие виды профилирования, касающиеся различных аспектов производительности, но все они включают запись последовательности интересующих событий, каждое из которых сопровождается трассировкой стека — списком функций, вызовы которых активны в момент события. Инструмент `go test` имеет встроенную поддержку нескольких видов профилирования.

Профиль процессора идентифицирует функции, выполнение которых требует больше всего процессорного времени. Текущий выполняющийся на каждом процессоре поток периодически прерывается операционной системой каждые несколько миллисекунд, и при каждом прерывании перед возобновлением нормальной работы выполняется запись одного события профилирования.

Профиль памяти определяет инструкции, ответственные за выделение наибольшего количества памяти. Выборки библиотеки профилирования вызывают внутренние процедуры выделения памяти так, что в среднем одно событие профиля записывается для каждых 512 Кбайт выделенной памяти.

Профиль блокировок выявляет операции, ответственные за наидлиннейшее блокирование go-подпрограмм, такие как системные вызовы, операции отправления и получения канала и захват блокировок. Библиотека профилирования записывает событие каждый раз, когда go-подпрограмма блокируется одной из этих операций.

Выполнить сбор информации профилей для тестируемого кода очень легко, включив один из перечисленных ниже флагов. Однако будьте осторожны при использовании более чем одного флага одновременно: сбор информации для одной разновидности профиля может исказить результаты для других.

```
$ go test -cpuprofile=cpu.out
$ go test -blockprofile=block.out
$ go test -memprofile=mem.out
```

Легко добавить поддержку профилирования и в не тестируемые программы, хотя детальная информации о том, как это делается, выглядит по-разному в зависимости от того, чем является приложение — коротким инструментом командной строки или длительно работающим серверным приложением. Профилирование особенно полезно в длительно работающих приложениях, поэтому возможности профилирования времени выполнения Go можно включать под контролем программиста с помощью API `runtime`.

Собрав нужную информацию, необходимо проанализировать ее с помощью инструмента `pprof`. Это стандартная часть дистрибутива Go, но поскольку это инструмент не повседневного использования, доступ к нему осуществляется косвенно с помощью команды `go tool pprof`. Он имеет множество возможностей и ключей, но в основном требуются только два аргумента — выполнимый файл, который создал профиль, и журнал профиля.

Чтобы сделать профилирование эффективным и сэкономить память, в журнал не включаются имена функций. Вместо этого функции идентифицируются по их адресам. Это значит, что для того, чтобы записи в журнале стали осмысленными, инструменту `pprof` необходим выполнимый файл. Хотя `go test` обычно уничтожает выполнимый файл после завершения теста, в случае профилирования выполнимый файл сохраняется как `foo.test`, где `foo` — это имя тестируемого пакета.

Приведенные ниже команды показывают, как выполнить сбор и отображение для простого профиля процессора. Мы выбрали одну из функций производительности пакета `net/http`. Обычно лучше профилировать конкретные функции, которые были специально созданы таким образом, чтобы быть репрезентативными для нагрузки, с которой придется иметь дело рабочей программе. Сравнительный анализ тестовых случаев почти никогда не является репрезентативным, и именно поэтому мы отключаем их с помощью фильтра `-run=NONE`.

```
$ go test -run=NONE -bench=ClientServerParallelTLS64 \
        -cpuprofile=cpu.log net/http
PASS
BenchmarkClientServerParallelTLS64-8    1000
        3141325 ns/op 143010 B/op 1747 allocs/op
ok    net/http    3.395s

$ go tool pprof -text -nodecount=10 ./http.test cpu.log
2570ms of 3590ms total (71.59%)
Dropped 129 nodes (cum <= 17.95ms)
Showing top 10 nodes out of 166 (cum >= 60ms)
    flat  flat%   sum%    cum   cum%
  1730ms 48.19% 48.19% 1750ms 48.75% crypto/elliptic.p256ReduceDegree
   230ms  6.41% 54.60%  250ms  6.96% crypto/elliptic.p256Diff
   120ms  3.34% 57.94%  120ms  3.34% math/big.addMulVVW
   110ms  3.06% 61.00%  110ms  3.06% syscall.Syscall
    90ms  2.51% 63.51% 1130ms 31.48% crypto/elliptic.p256Square
    70ms  1.95% 65.46%  120ms  3.34% runtime.scanobject
    60ms  1.67% 67.13%  830ms 23.12% crypto/elliptic.p256Mul
    60ms  1.67% 68.80%  190ms  5.29% math/big.nat.montgomery
    50ms  1.39% 70.19%   50ms  1.39% crypto/elliptic.p256ReduceCarry
    50ms  1.39% 71.59%   60ms  1.67% crypto/elliptic.p256Sum
```

Флаг `-text` указывает формат вывода, в данном случае — в виде текстовой таблицы с одной строкой на каждую функцию, отсортированную в порядке "горячести" функций, — те, которые потребляют больше процессорного времени, идут первыми. Флаг `-nodecount=10` ограничивает результат десятью строками. Для выясне-

ния причин больших проблем производительности этого текстового формата должно быть достаточно.

Этот профиль говорит нам о том, что криптография на основе эллиптических кривых имеет важное значение для выполнения этой конкретной проверки производительности HTTPS. Напротив, если в профиле доминируют функции выделения памяти из пакета `runtime`, наилучшей оптимизацией может оказаться сокращение потребления памяти.

Для более тонких проблем, может быть, лучше воспользоваться одним из графических выводов `pprof`. Они требуют пакета GraphViz, который может быть загружен с сайта по адресу `www.graphviz.org`. Флаг `-web` выводит ориентированный граф функций программы, аннотированный их используемым процессорным временем и цветом для указания "горячих" функций.

Здесь мы только бегло ознакомились с инструментами профилирования. Чтобы получить больше информации, обратитесь к статье *Profiling Go Programs* в блоге Go Blog.

11.6. Функции-примеры

Третий вид функций, обрабатываемых командой `go test` специальным образом, — функции-примеры, имя которых начинается с `Example`. Они не имеют ни параметров, ни результатов. Вот пример такой функции для `IsPalindrome`:

```
func ExampleIsPalindrome() {
    fmt.Println(IsPalindrome("A man, a plan, a canal: Panama"))
    fmt.Println(IsPalindrome("palindrome"))
    // Output:
    // true
    // false
}
```

Функции-примеры служат трем целям. Основная из них — документация: хороший пример может быть более сжатым или интуитивно понятным средством передачи информации о поведении библиотечной функции, чем ее словесное описание, особенно при использовании в качестве напоминания или краткой справки. Пример может также продемонстрировать взаимодействие между несколькими типами и функциями, принадлежащими одному API, тогда как словесная документация всегда оказывается привязанной к одному объекту, такому как тип, объявление функции или пакет в целом. И, в отличие от примеров в комментариях, функции-примеры являются реальным кодом Go, проверяемым во время компиляции, так что они не становятся устаревшими по мере развития рабочего кода.

Основываясь на суффиксе `Example`, сервер документации на базе веб `godoc` связывает функции-примеры с функцией или пакетом, который они иллюстрируют, так что `ExampleIsPalindrome` будет отображаться в документации для функции `IsPalindrome`, и в качестве функции с именем `Example` будет связан с пакетом `word` в целом.

Второе предназначение функций-примеров заключается в том, что примеры являются выполнимыми тестами, запускаемыми командой `go test`. Если в функции-примере содержится завершающий комментарий `// Output:`, как показано выше, тест-драйвер выполнит эту функцию и проверит, выводит ли она в стандартный поток вывода текст, указанный в комментарии.

Третья цель примера — практические эксперименты. Сервер `godoc` по адресу `golang.org` использует Go Playground, чтобы позволить пользователю редактировать и запускать каждую функцию-пример из веб-браузера, как показано на рис. 11.4. Часто это самый быстрый способ "пощупать" определенную функцию или возможность языка.

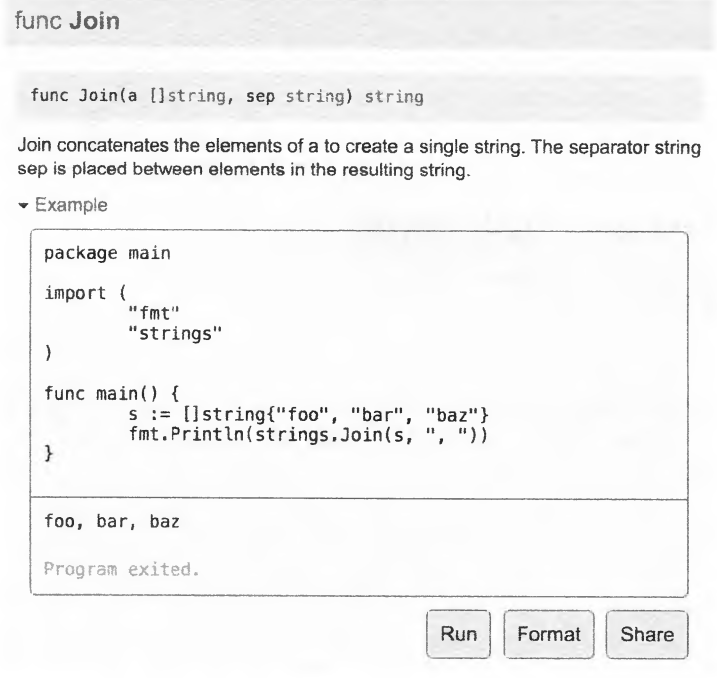


Рис. 11.4. Интерактивный пример `strings.Join` в `godoc`

В последних двух главах книги описаны пакеты `reflect` и `unsafe`, которые регулярно используют лишь несколько программистов на Go (и еще меньшему количеству они реально *нужны*). Если вы пока еще не написали каких-либо особо существенных программ на Go, сейчас самое время заняться этим.

12

Рефлексия

Go предоставляет механизм для обновления переменных и проверки их значений во время выполнения, для вызова их методов и для применения операций, присущих их представлениям, и все это без знания об их типах во время компиляции. Этот механизм называется *рефлексией* (reflection). Рефлексия также позволяет рассматривать сами типы как значения первого класса.

В этой главе мы будем изучать возможности рефлексии Go, чтобы увидеть, как они увеличивают выразительность языка, и в частности, какое решающее значение они имеют для реализации двух важных API: форматирования строк, предоставляемого `fmt`, и кодирования протоколов, предоставляемого такими пакетами, как `encoding/json` и `encoding/xml`. Рефлексия также имеет важное значение для механизма шаблонов, предоставляемого пакетами `text/template` и `html/template`, которые мы видели в разделе 4.6. Однако рефлексия — слишком сложный для понимания предмет, не предназначенный для повседневного использования, так что, хотя эти пакеты и реализованы с использованием рефлексии, в своих API они не предоставляют никакой рефлексии.

12.1. Почему рефлексия?

Иногда нужно написать функцию, способную работать одинаково со значениями типов, которые не удовлетворяют общему интерфейсу, не имеют общего представления или не существуют в то время, когда мы разрабатываем данную функцию (все три варианта могут осуществляться одновременно).

Знакомым примером является логика форматирования в `fmt.Fprintf`, которая позволяет вывести разумное представление произвольного значения любого типа, в том числе определенного пользователем. Попробуем реализовать функцию, подобную ей, используя все свои знания языка программирования Go. Для простоты наша функция будет принимать один аргумент и возвращать результат в виде строки, как делает `fmt.Sprint`, так что мы назовем ее `Sprint`.

Мы начинаем с выбора типа, который выясняет, определяет ли аргумент метод `String`, и вызывает его, если это так. Затем мы добавляем варианты, которые проверяют, не совпадает ли динамический тип значения с одним из фундаментальных

типов — string, int, bool и т.д., — и выполняют соответствующие операции форматирования для каждого конкретного случая.

```
func Sprint(x interface{}) string {
    type stringer interface {
        String() string
    }
    switch x := x.(type) {
    case stringer:
        return x.String()
    case string:
        return x
    case int:
        return strconv.Itoa(x)
        // ... аналогичные case для int16, uint32 и т.д. ...
    case bool:
        if x {
            return "true"
        }
        return "false"
    default:
        // массив, канал, функция, отображение,
        // указатель, срез, структура...
        return "???"
    }
}
```

Но как быть с другими типами, такими как `[]float64`, `map[string][]string` и т.д.? Мы могли бы добавить больше вариантов `case`, однако количество таких типов бесконечно. А как быть с именованными типами, такими как `url.Values`? Даже если в выборе типа имеется `case` для его базового типа `map[string][]string`, он не будет соответствовать `url.Values`, поскольку эти два типа не идентичны, а выбор типа не может включать `case` для каждого типа наподобие `url.Values`, поскольку это потребовало зависимости библиотеки от ее клиентов.

Без способа проверки представления значений неизвестных типов мы быстро зашли в тупик. Что нам нужно, так это рефлексия.

12.2. `reflect.Type` и `reflect.Value`

Рефлексия обеспечивается пакетом `reflect`. Он определяет два важных типа, `Type` и `Value`. `Type` представляет собой тип Go. Это интерфейс со многими методами для того, чтобы можно было различать типы и проверять их компоненты, такие как поля структуры или параметры функции. Единственная реализация `reflect.Type` представляет собой дескриптор типа (раздел 7.5), ту же сущность, что и идентифицирующая динамический тип значения интерфейса.

Функция `reflect.TypeOf` принимает любой `interface{}` и возвращает его динамический тип как `reflect.Type`:

```
t := reflect.TypeOf(3)   // reflect.Type
fmt.Println(t.String())  // "int"
fmt.Println(t)           // "int"
```

Показанный выше вызов `TypeOf(3)` присваивает значение `3` параметру `interface{}`. Вспомните из раздела 7.5, что присваивание конкретного значения типу интерфейса выполняет неявное преобразование интерфейса, которое создает значение интерфейса, состоящее из двух компонентов: его *динамический тип* является типом операнда (`int`), а его *динамическое значение* равно значению операнда (`3`).

Поскольку `reflect.TypeOf` возвращает динамический тип значения интерфейса, этот вызов всегда возвращает конкретный тип. Так что, например, приведенный ниже код выводит `"*os.File"`, а не `"io.Writer"`. Позже мы увидим, что `reflect.Type` в состоянии представлять и интерфейсные типы:

```
var w io.Writer = os.Stdout
fmt.Println(reflect.TypeOf(w)) // "*os.File"
```

Обратите внимание, что `reflect.Type` удовлетворяет интерфейсу `fmt.Stringer`. Поскольку знать динамический тип значения интерфейса полезно для отладки и записи в журнал, `fmt.Printf` предоставляет соответствующее сокращение `%T`, которое использует `reflect.TypeOf` во внутренней реализации:

```
fmt.Printf("%T\n", 3) // "int"
```

Другим важным типом в пакете `reflect` является `Value`. `reflect.Value` может хранить значение любого типа. Функция `reflect.ValueOf` принимает любой `interface{}` и возвращает значение `reflect.Value`, содержащее динамическое значение интерфейса. Как и в случае `reflect.TypeOf`, результат `reflect.ValueOf` всегда конкретен, но `reflect.Value` может хранить и значения интерфейса:

```
v := reflect.ValueOf(3) // reflect.Value
fmt.Println(v)          // "3"
fmt.Printf("%v\n", v)   // "3"
fmt.Println(v.String()) // Примечание: "<int Value>"
```

Как и `reflect.Type`, `reflect.Value` также удовлетворяет интерфейсу `fmt.Stringer`, но если только `Value` не содержит строку, результат метода `String` выдает тип. Поэтому используйте символы преобразования `%v` из пакета `fmt`, которые трактуют `reflect.Value` особым образом.

Вызов метода `Type` для `Value` возвращает его тип как `reflect.Type`:

```
t := v.Type()           // reflect.Type
fmt.Println(t.String()) // "int"
```

Обратной к `reflect.ValueOf` операцией является метод `reflect.Value.Interface`. Он возвращает `interface{}`, хранящий то же самое конкретное значение, что и `reflect.Value`:

```
v := reflect.ValueOf(3) // reflect.Value
x := v.Interface()      // interface{}
i := x.(int)            // int
fmt.Printf("%d\n", i)   // "3"
```

И `reflect.Value`, и `interface{}` могут хранить произвольные значения. Разница в том, что пустой интерфейс скрывает представление и внутренние операции значения, которое он хранит, и не предоставляет ни один из его методов, так что, если только мы не знаем его динамический тип и не используем декларацию типа, чтобы заглянуть внутрь него (как мы это делали выше), мы мало что можем сделать с этим значением. В противоположность этому `Value` имеет много методов для работы с его содержимым, независимо от типа. Давайте используем их при нашей второй попытке написать обобщенную функцию форматирования, которую мы назовем `format.Any`.

Вместо выбора типа мы воспользуемся методом `reflect.Value` под названием `Kind` для того, чтобы различать типы. Хотя существует бесконечно много типов, имеется лишь конечное количество *разновидностей* типов: фундаментальные типы `Bool`, `String` и все числовые типы; составные типы `Array` и `Struct`; ссылочные типы `Chan`, `Func`, `Ptr`, `Slice` и `Map`; типы `Interface`; и наконец `Invalid`, означающий отсутствие значения вообще. (Нулевое значение `reflect.Value` имеет разновидность `Invalid`.)

gopl.io/ch12/format

```
package format

import (
    "reflect"
    "strconv"
)

// Any форматирует любое значение в виде строки.
func Any(value interface{}) string {
    return formatAtom(reflect.ValueOf(value))
}

// formatAtom форматирует значение без
// исследования его внутренней структуры.
func formatAtom(v reflect.Value) string {
    switch v.Kind() {
    case reflect.Invalid:
        return "invalid"
    case reflect.Int, reflect.Int8, reflect.Int16,
        reflect.Int32, reflect.Int64:
        return strconv.FormatInt(v.Int(), 10)
    case reflect.Uint, reflect.Uint8, reflect.Uint16,
        reflect.Uint32, reflect.Uint64, reflect.Uintptr:
        return strconv.FormatUint(v.Uint(), 10)
    // ... типы с плавающей точкой и комплексные
    // числа опущены для краткости...
```

```
    case reflect.Bool:
        return strconv.FormatBool(v.Bool())
    case reflect.String:
        return strconv.Quote(v.String())
    case reflect.Chan, reflect.Func, reflect.Ptr,
        reflect.Slice, reflect.Map:
        return v.Type().String() + " 0x" +
            strconv.FormatUint(uint64(v.Pointer()), 16)
    default: // reflect.Array, reflect.Struct, reflect.Interface
        return v.Type().String() + " value"
    }
}
```

До сих пор наша функция рассматривала каждое значение как неделимый объект без внутренней структуры — отсюда и название `formatAtom`. Для интерфейсов и составных типов (структур и массивов) она выдает только *тип* значения, а для ссылочных типов (каналов, функций, указателей, срезов и отображений) выводит тип и адрес ссылки в шестнадцатеричном формате. Это не идеальное решение, но все же значительное улучшение, а поскольку `Kind` связан только с базовым представлением, `format.Any` работает и для именованных типов, например:

```
var x int64 = 1
var d time.Duration = 1 * time.Nanosecond
fmt.Println(format.Any(x))          // "1"
fmt.Println(format.Any(d))          // "1"
fmt.Println(format.Any([]int64{x})) // "[]int64 0x8202b87b0"
fmt.Println(format.Any([]time.Duration{d}))
                                    // "[]time.Duration 0x8202b87e0"
```

12.3. Рекурсивный вывод значения

Давайте теперь рассмотрим, как усовершенствовать отображение составных типов. Вместо того чтобы пытаться в точности копировать `fmt.Sprint`, мы будем строить вспомогательную функцию для отладки с названием `Display`, которая для данного значения произвольно сложного значения `x` выводит полную структуру этого значения, помечая каждый элемент путем, на котором он был найден. Начнем с примера:

```
e, _ := eval.Parse("sqrt(A / pi)")
Display("e", e)
```

В приведенном выше вызове аргументом `Display` является синтаксическое дерево, полученное вычислителем выражений из раздела 7.9. Вывод `Display` показан ниже:

```
Display e (eval.call):
e.fn = "sqrt"
e.args[0].type = eval.binary
e.args[0].value.op = 47
```

```
e.args[0].value.x.type = eval.Var
e.args[0].value.x.value = "A"
e.args[0].value.y.type = eval.Var
e.args[0].value.y.value = "pi"
```

Где это возможно, вы должны избегать демонстрации рефлексии в API пакета. Мы определим неэкспортируемую функцию `display`, которая выполняет всю реальную работу, и экспортируем `Display`, простую оболочку вокруг нее, которая принимает параметр `interface{}`:

gopl.io/ch12/display

```
func Display(name string, x interface{}) {
    fmt.Printf("Display %s (%T):\n", name, x)
    display(name, reflect.ValueOf(x))
}
```

В функции `display` мы используем функцию `formatAtom`, которую мы определили ранее для вывода элементарных значений — фундаментальных типов, функций и каналов, — но мы используем методы `reflect.Value` для рекурсивного вывода каждого компонента более сложного типа. По мере рекурсивного спуска строка `path`, которая сперва описывает начальное значение (например, `"e"`), будет расширяться для указания того, как мы достигли текущего значения (например, `"e.args[0].value"`).

Поскольку мы больше не притворяемся, что реализуем `fmt.Sprint`, воспользуемся пакетом `fmt` для того, чтобы наш пример оставался коротким:

```
func display(path string, v reflect.Value) {
    switch v.Kind() {
    case reflect.Invalid:
        fmt.Printf("%s = invalid\n", path)
    case reflect.Slice, reflect.Array:
        for i := 0; i < v.Len(); i++ {
            display(fmt.Sprintf("%s[%d]", path, i), v.Index(i))
        }
    case reflect.Struct:
        for i := 0; i < v.NumField(); i++ {
            fieldPath := fmt.Sprintf("%s.%s", path,
                                     v.Type().Field(i).Name)
            display(fieldPath, v.Field(i))
        }
    case reflect.Map:
        for _, key := range v.MapKeys() {
            display(fmt.Sprintf("%s[%s]", path,
                formatAtom(key)), v.MapIndex(key))
        }
    case reflect.Ptr:
        if v.IsNil() {
            fmt.Printf("%s = nil\n", path)
        } else {
```

```
            display(fmt.Sprintf("(*%s)", path), v.Elem())
        }
    case reflect.Interface:
        if v.IsNil() {
            fmt.Printf("%s = nil\n", path)
        } else {
            fmt.Printf("%s.type = %s\n", path, v.Elem().Type())
            display(path+".value", v.Elem())
        }
    default: // basic types, channels, funcs
        fmt.Printf("%s = %s\n", path, formatAtom(v))
    }
}
```

Давайте рассмотрим все `case` по порядку.

Срезы и массивы. Логика одинакова для них обоих. Метод `Len` возвращает количество элементов среза или массива, а `Index(i)` выбирает элемент с индексом `i`, также как `reflect.Value`; если `i` выходит за границы, возникает аварийная ситуация. Это аналогично встроенным операциям `len(a)` и `a[i]` для последовательностей. Функция `display` рекурсивно вызывает саму себя для каждого элемента последовательности, добавляя к пути индекс `"[i]"`.

Хотя `reflect.Value` имеет много методов, только некоторые из них могут быть безопасно вызваны для любого заданного значения. Например, метод `Index` может быть вызван для значений разновидностей `Slice`, `Array` и `String`, но приводит к аварийной ситуации для всех прочих разновидностей.

Структуры. Метод `NumField` сообщает о количестве полей в структуре, а `Field(i)` возвращает значение *i*-го поля в виде `reflect.Value`. Список полей включает анонимные поля. Для добавления записи поля `".f"` к пути мы должны получить `reflect.Type` для структуры и обратиться к имени *i*-го поля.

Отображения. Метод `MapKeys` возвращает срез значений `reflect.Value`, по одному для каждого ключа отображения. Как обычно при итерациях по отображению, порядок итераций не определен. `MapIndex(key)` возвращает значение, соответствующее `key`. Мы добавляем к пути обозначения индекса `"[key]"`. (Здесь мы срезаем угол. Тип ключа отображения не ограничен типами, которые лучше обрабатываются функцией `formatAtom`; массивы, структуры и интерфейсы также могут быть корректными ключами отображения. Расширение этого варианта так, чтобы ключ выводился полностью, является заданием упражнения 12.1.)

Указатели. Метод `Elem` возвращает переменную, на которую указывает указатель, вновь как `reflect.Value`. Эта операция безопасна, даже если значение указателя — `nil`. В последнем случае результат будет иметь разновидность `Invalid`, но мы используем `IsNil`, чтобы обнаруживать нулевые указатели явно, так что мы в состоянии вывести более подходящее случаю сообщение. Во избежание неоднозначностей мы добавляем в путь префикс `"*"` и скобки.

Интерфейсы. И вновь мы используем `IsNil` для проверки, не равен ли интерфейс `nil`, и если нет, мы получаем динамическое значение с использованием `v.Elem()` и выводим его тип и значение.

Теперь, когда наша функция `Display` завершена, давайте посмотрим ее в работе. Приведенный ниже тип `Movie` является небольшой вариацией типа из раздела 4.5:

```
type Movie struct {
    Title, Subtitle  string
    Year             int
    Color            bool
    Actor            map[string]string
    Oscars           []string
    Sequel           *string
}
```

Давайте объявим значение этого типа и посмотрим, что с ним сделает функция `Display`:

```
strangelove := Movie{
    Title: "Dr. Strangelove",
    Subtitle: "How I Learned to Stop Worrying and Love the Bomb",
    Year: 1964,
    Color: false,
    Actor: map[string]string{
        "Dr. Strangelove":            "Peter Sellers",
        "Grp. Capt. Lionel Mandrake": "Peter Sellers",
        "Pres. Merkin Muffley":       "Peter Sellers",
        "Gen. Buck Turgidson":        "George C. Scott",
        "Brig. Gen. Jack D. Ripper":  "Sterling Hayden",
        `Maj. T.J. "King" Kong`:      "Slim Pickens",
    },
    Oscars: []string{
        "Best Actor (Nomin.)",
        "Best Adapted Screenplay (Nomin.)",
        "Best Director (Nomin.)",
        "Best Picture (Nomin.)",
    },
}
```

Вызов `Display("strangelove",strangelove)` выводит следующее:

```
Display strangelove (display.Movie):
strangelove.Title = "Dr. Strangelove"
strangelove.Subtitle = "How I Learned to Stop Worrying and Love the Bomb"
strangelove.Year = 1964
strangelove.Color = false
strangelove.Actor["Gen. Buck Turgidson"] = "George C. Scott"
strangelove.Actor["Brig. Gen. Jack D. Ripper"] = "Sterling Hayden"
strangelove.Actor["Maj. T.J. \"King\" Kong"] = "Slim Pickens"
strangelove.Actor["Dr. Strangelove"] = "Peter Sellers"
```

```
strangelove.Actor["Grp. Capt. Lionel Mandrake"] = "Peter Sellers"
strangelove.Actor["Pres. Merkin Muffley"] = "Peter Sellers"
strangelove.Oscars[0] = "Best Actor (Nomin.)"
strangelove.Oscars[1] = "Best Adapted Screenplay (Nomin.)"
strangelove.Oscars[2] = "Best Director (Nomin.)"
strangelove.Oscars[3] = "Best Picture (Nomin.)"
strangelove.Sequel = nil
```

Мы можем использовать `Display` для вывода информации о внутреннем устройстве библиотечных типов, таких как `*os.File`:

```
Display("os.Stderr", os.Stderr)
// Output:
// Display os.Stderr (*os.File):
// (*(*os.Stderr).file).fd = 2
// (*(*os.Stderr).file).name = "/dev/stderr"
// (*(*os.Stderr).file).nepipe = 0
```

Обратите внимание, что даже неэкспортируемые поля являются видимыми для рефлексии. Учтите, что конкретный вывод данного примера может отличаться в разных платформах и даже изменяться со временем, по мере развития библиотек. (Данные поля потому и являются закрытыми!) Мы можем даже применить `Display` к `reflect.Value` и посмотреть на внутреннее представление дескриптора типа для `*os.File`. Результат вызова `Display("rV",reflect.ValueOf(os.Stderr))` приведен ниже, хотя, конечно, он может варьироваться:

```
Display rV (reflect.Value):
(*rV.typ).size = 8
(*rV.typ).hash = 871609668
(*rV.typ).align = 8
(*rV.typ).fieldAlign = 8
(*rV.typ).kind = 22
(*(*rV.typ).string) = "*os.File"
(*(*(*rV.typ).uncommonType).methods[0].name) = "Chdir"
(*(*(*(*rV.typ).uncommonType).methods[0].mtyp).string)
                = "func() error"
(*(*(*(*rV.typ).uncommonType).methods[0].typ).string)
                = "func(*os.File) error"
...
```

Обратите внимание на различие этих двух примеров:

```
var i interface{} = 3
Display("i", i)
// Output:
// Display i (int):
// i = 3
Display("&i", &i)
// Output:
// Display &i (*interface {}):
```

```
// (*&i).type = int
// (*&i).value = 3
```

В первом примере `Display` осуществляет вызов `reflect.ValueOf(i)`, который возвращает значение разновидности `Int`. Как мы упоминали в разделе 12.2, `reflect.ValueOf` всегда возвращает `Value` конкретного типа, поскольку извлекает содержимое значения интерфейса.

Во втором примере `Display` осуществляет вызов `reflect.ValueOf(&i)`, который возвращает указатель на `i`, разновидности `Ptr`. Вариант `case` для `Ptr` вызывает `Elem` для этого значения, и этот вызов возвращает значение, представляющее саму *переменную* `i`, разновидности `Interface`. Полученное косвенно `Value`, такое как только что упомянутое, может представлять любое значение, включая интерфейсы. Функция `display` рекурсивно вызывает сама себя и на этот раз выводит отдельные компоненты для динамических типа и значения интерфейса.

В текущей реализации `Display` никогда не завершается при обнаружении цикла в графе объектов, такого как этот связанный список, который "кусает себя за собственный хвост":

```
// Структура, указывающая сама на себя
type Cycle struct{ Value int; Tail *Cycle }
var c Cycle
c = Cycle{42, &c}
Display("c", c)
```

`Display` выводит следующие постоянно растущие строки:

```
Display c (display.Cycle):
c.Value = 42
(*c.Tail).Value = 42
(*(*c.Tail).Tail).Value = 42
(*(*(*c.Tail).Tail).Tail).Value = 42
... до бесконечности ...
```

Многие программы Go содержат по крайней мере некоторые циклические данные. Сделать функцию `Display` успешно обрабатывающей такие циклы — сложная задача, требующая дополнительной записи набора ссылок, по которым к настоящему времени был выполнен проход, а это слишком дорого. Общее решение требует применения языковых возможностей из пакета `unsafe`, с которым мы познакомимся в разделе 13.3.

Циклы создают меньше проблем для `fmt.Sprint`, потому что эта функция редко пытается вывести полную структуру. Например, встретив указатель, она прекращает рекурсию, выводя числовое значение этого указателя. Она может оказаться в тупике, пытаться вывести срез или отображение, содержащее себя в качестве элемента, но такие редкие случаи не оправдывают значительные дополнительные действия по обработке циклов.

Упражнение 12.1. Расширьте `Display` так, чтобы эта функция могла выводить отображения, ключами которых являются структуры или массивы.

Упражнение 12.2. Сделайте функцию `display` безопасной для применения с циклическими структурами данных путем ограничения количества шагов, которые она выполняет, перед тем как прервать рекурсию. (В разделе 13.3 вы познакомитесь с другим способом обнаружения циклов.)

12.4. Пример: кодирование S-выражений

`Display` является отладочной подпрограммой для вывода структурированных данных, но не менее интересна возможность ее применения для кодирования произвольных объектов Go в виде переносимых сообщений для межпроцессного взаимодействия.

Как мы видели в разделе 4.5, стандартная библиотека Go поддерживает множество форматов, включая XML, JSON и ASN.1. Еще одной все еще широко используемой записью являются *S-выражения*, синтаксис Lisp. В отличие от других обозначений, S-выражения не поддерживаются стандартной библиотекой Go, не в последнюю очередь потому, что у них так и нет общепринятого определения, несмотря на несколько попыток стандартизации и существование многих реализаций.

В этом разделе мы определим пакет, который кодирует произвольные объекты Go с использованием S-выражении и который поддерживает следующие конструкции:

```
42          целое число
"hello"     строка (в кавычках в стиле Go)
foo         символ (имя без кавычек)
(1 2 3)     список (нуль или несколько элементов в скобках)
```

Булевы значения традиционно кодируются с помощью символа `t` для `true` и пустого списка `()` или имени `nil` для `false`, но для простоты наша реализация их игнорирует. Она также игнорирует каналы и функции, так как их состояние является непрозрачным для рефлексии. А еще она игнорирует действительные и комплексные числа с плавающей точкой и интерфейсы. Добавление поддержки для них — задание упражнение 12.3.

Мы будем кодировать типы Go с помощью S-выражений следующим образом. Целые числа и строки кодируются очевидным способом. Нулевые значения кодируются с помощью символа `nil`. Массивы и срезы кодируются с помощью обозначения списка.

Структуры кодируются как список полей, при этом каждое поле представляет собой два элемента списка, в котором первый элемент (символ) — это имя поля, а вторым элементом является значение поля. Отображения также кодируются в виде списка пар, при этом каждая пара содержит ключ и значение одной записи отображения. Традиционно S-выражения представляют собой списки пар "ключ/значение", использующие для каждой пары ячейки `(key.value)`, а не двухэлементный список, но для упрощения декодирования мы будем игнорировать такую запись.

Кодирование осуществляется одной рекурсивной функцией `encode`, показанной ниже. Ее структура, по существу, представляет собой то же самое, что и структура функции `Display` из предыдущего раздела:

```
gopl.io/ch12/sexpr
func encode(buf *bytes.Buffer, v reflect.Value) error {
    switch v.Kind() {
    case reflect.Invalid:
        buf.WriteString("nil")
    case reflect.Int, reflect.Int8, reflect.Int16,
        reflect.Int32, reflect.Int64:
        fmt.Fprintf(buf, "%d", v.Int())
    case reflect.Uint, reflect.Uint8, reflect.Uint16,
        reflect.Uint32, reflect.Uint64, reflect.Uintptr:
        fmt.Fprintf(buf, "%d", v.Uint())
    case reflect.String:
        fmt.Fprintf(buf, "%q", v.String())
    case reflect.Ptr:
        return encode(buf, v.Elem())
    case reflect.Array, reflect.Slice: // (value ...)
        buf.WriteByte('(')
        for i := 0; i < v.Len(); i++ {
            if i > 0 {
                buf.WriteByte(' ')
            }
            if err := encode(buf, v.Index(i)); err != nil {
                return err
            }
        }
        buf.WriteByte(')')
    case reflect.Struct: // ((name value) ...)
        buf.WriteByte('(')
        for i := 0; i < v.NumField(); i++ {
            if i > 0 {
                buf.WriteByte(' ')
            }
            fmt.Fprintf(buf, "(%s ", v.Type().Field(i).Name)
            if err := encode(buf, v.Field(i)); err != nil {
                return err
            }
            buf.WriteByte(')')
        }
        buf.WriteByte(')')
    case reflect.Map: // ((key value) ...)
        buf.WriteByte('(')
        for i, key := range v.MapKeys() {
            if i > 0 {
                buf.WriteByte(' ')
            }
            buf.WriteByte('(')
            if err := encode(buf, key); err != nil {
                return err
```

```
            }
            buf.WriteByte(' ')
            if err := encode(buf, v.MapIndex(key)); err != nil {
                return err
            }
            buf.WriteByte(')')
        }
        buf.WriteByte(')')
    default: // float, complex, bool, chan, func, interface
        return fmt.Errorf("неподдерживаемый тип: %s", v.Type())
    }
    return nil
}
```

Функция `Marshal` "оборачивает" кодировщик в API, подобный API других пакетов `encoding/...`:

```
// Marshal кодирует значение Go в виде S-выражения.
func Marshal(v interface{}) ([]byte, error) {
    var buf bytes.Buffer
    if err := encode(&buf, reflect.ValueOf(v)); err != nil {
        return nil, err
    }
    return buf.Bytes(), nil
}
```

Вот как выглядит вывод функции `Marshal`, примененной к переменной `strangelove` из раздела 12.3:

```
((Title "Dr. Strangelove") (Subtitle "How I Learned to Stop Worrying
and Love the Bomb") (Year 1964) (Actor (("Grp. Capt. Lionel Mandrake"
 "Peter Sellers") ("Pres. Merkin Muffley" "Peter Sellers") ("Gen. Buc
k Turgidson" "George C. Scott") ("Brig. Gen. Jack D. Ripper" "Sterlin
g Hayden") ("Maj. T.J. \"King\" Kong" "Slim Pickens") ("Dr. Strangelo
ve" "Peter Sellers"))) (Oscars("Best Actor (Nomin.)" "Best Adapted Sc
reenplay (Nomin.)" "Best Director (Nomin.)" "Best Picture (Nomin.)"))
 (Sequel nil))
```

Весь вывод представляет собой одну строку с минимальными пробелами, которую очень трудно читать. Вот как выглядит тот же вывод, отформатированный вручную согласно соглашениям S-выражений. Написание красивого вывода S-выражений остается читателям в качестве (сложного) упражнения; загрузка с адреса `gopl.io` включает простую версию.

```
((Title "Dr. Strangelove")
 (Subtitle "How I Learned to Stop Worrying and Love the Bomb")
 (Year 1964)
 (Actor (("Grp. Capt. Lionel Mandrake" "Peter Sellers")
         ("Pres. Merkin Muffley" "Peter Sellers")
         ("Gen. Buck Turgidson" "George C. Scott")
```

```
        ("Brig. Gen. Jack D. Ripper" "Sterling Hayden")
        ("Maj. T.J. \"King\" Kong" "Slim Pickens")
        ("Dr. Strangelove" "Peter Sellers")))
 (Oscars ("Best Actor (Nomin.)"
          "Best Adapted Screenplay (Nomin.)"
          "Best Director (Nomin.)"
          "Best Picture (Nomin.)"))
 (Sequel nil))
```

Так же, как и функции `fmt.Print`, `json.Marshal` и `Display`, `sexpr.Marshal` зацикливается при передаче ей циклических данных.

В разделе 12.6 мы наметим реализацию соответствующей функции декодирования S-выражений, но прежде чем перейти к нему, нужно понять, как рефлексия может использоваться для обновления переменных программы.

Упражнение 12.3. Реализуйте пропущенные случаи в функции `encode`. Кодируйте булевы значения как `t` и `nil`, числа с плавающей точкой с помощью обычной записи Go, а комплексные числа наподобие $1+2i$ — как `#C(1.0 2.0)`. Интерфейсы можно кодировать как пары, состоящие из имени типа и значения, например `("[]int" (1 2 3))`, но учтите, что такая кодировка неоднозначна: метод `reflect.Type.String` может возвращать такую же строку для различных типов.

Упражнение 12.4. Модифицируйте `encode` так, чтобы вывод S-выражений осуществлялся в удобочитаемом виде, приведенном выше.

Упражнение 12.5. Адаптируйте функцию `encode` так, чтобы она выводила запись в формате JSON вместо S-выражений. Протестируйте свой кодировщик с помощью стандартного декодировщика `json.Unmarshal`.

Упражнение 12.6. Адаптируйте функцию `encode` так, чтобы в качестве оптимизации она не кодировала поля, значения которых являются нулевыми значениями их типов.

Упражнение 12.7. Создайте потоковое API для декодировщика S-выражений, следуя стилю `json.Decoder` (раздел 4.5).

12.5. Установка переменных с помощью `reflect.Value`

Пока что рефлексия применялась только для *интерпретации* значений в программе различными способами. Суть данного раздела в том, чтобы их *изменять*.

Вспомним, что одни выражения Go наподобие `x`, `x.f[1]` или `*p` обозначают переменные, а другие, такие как `x + 1` или `f(2)`, — нет. Переменная представляет собой *адресуемое* местоположение в памяти, которое содержит значение, и это значение может быть изменено с использованием адреса переменной.

Аналогичное различие применяется и к `reflect.Value`. Одни из них являются доступными, другие — нет. Рассмотрим следующие объявления:

```
x := 2                   // значение   тип  переменная?
a := reflect.ValueOf(2)  //    2      int     no
```

```
b := reflect.ValueOf(x)  //    2      int     no
c := reflect.ValueOf(&x) //   &x     *int     no
d := c.Elem()            //    2      int     yes (x)
```

Значение в a не адресуемое. Это всего лишь копия целого числа 2. То же самое относится и к b. Значение в c также неадресуемое, будучи копией значения указателя &x. Фактически значения reflect.Value, возвращенные reflect.ValueOf(x), не являются адресуемыми. Но d, производное от c путем разыменования указателя внутри него, ссылается на переменную и, таким образом, является адресуемым. Мы можем использовать этот подход, вызывая reflect.ValueOf(&x).Elem() для получения значение адресуемого Value для любой переменной x.

Мы можем запросить у reflect.Value, является ли значение адресуемым, с помощью метода CanAddr:

```
fmt.Println(a.CanAddr()) // "false"
fmt.Println(b.CanAddr()) // "false"
fmt.Println(c.CanAddr()) // "false"
fmt.Println(d.CanAddr()) // "true"
```

Мы получаем адресуемое reflect.Value при косвенном обращении через указатель, даже если начинаем с неадресуемого Value. Все обычные правила адресации имеют аналоги для рефлексии. Например, поскольку выражение индексации среза e[i] неявно использует указатель, оно адресуемо, даже если выражение e таковым не является. По аналогии reflect.ValueOf(e).Index(i) ссылается на переменную и, таким образом, является адресуемым, даже если reflect.ValueOf(e) таковым не является.

Для восстановления переменной из адресуемого reflect.Value требуются три шага. Сначала мы вызываем Addr(), получая Value, которое содержит указатель на переменную. Далее для этого Value мы вызываем Interface(), который возвращает значение interface{}, содержащее указатель. Наконец, если мы знаем тип переменной, мы можем использовать декларацию типа для извлечения содержимого интерфейса как обычного указателя. Затем мы можем обновить переменную через этот указатель:

```
x := 2
d := reflect.ValueOf(&x).Elem()   // d ссылается на переменную x
px := d.Addr().Interface().(*int) // px := &x
*px = 3                           // x = 3
fmt.Println(x)                    // "3"
```

Мы также можем обновить переменную, на которую ссылается адресуемое значение reflect.Value, непосредственно, без применения указателя, с помощью вызова метода reflect.Value.Set:

```
d.Set(reflect.ValueOf(4))
fmt.Println(x)              // "4"
```

Методом Set во время выполнения проводятся те же проверки доступности, которые обычно выполняются с помощью компилятора. Выше переменная и значение

имеют тип int, но если бы переменная была типа int64, возникла бы аварийная ситуация, поэтому очень важно убедиться, что данное значение присваиваемо типу переменной:

```
d.Set(reflect.ValueOf(int64(5))) // Аварийная ситуация:
                                 // int64 не присваиваемо типу int
```

И конечно же, вызов Set для неадресуемого reflect.Value также приведет к аварийной ситуации:

```
x := 2
b := reflect.ValueOf(x)
b.Set(reflect.ValueOf(3)) // Аварийная ситуация:
                          // неадресуемое значение передано Set
```

Имеются варианты Set, специализированные для некоторых групп фундаментальных типов: SetInt, SetUint, SetString, SetFloat и т.д.:

```
d := reflect.ValueOf(&x).Elem()
d.SetInt(3)
fmt.Println(x)      // "3"
```

В некоторых отношениях эти методы являются более снисходительными. SetInt, например, будет успешным, если тип переменной представляет собой некоторую разновидность знакового целого числа или даже именованный тип, базовым типом которого является знаковое целое число, и если значение слишком велико, оно будет молча усечено. Но будьте внимательны: вызов SetInt для reflect.Value, которое ссылается на переменную interface{}, приведет к аварийной ситуации, даже несмотря на успешность Set:

```
x := 1
rx := reflect.ValueOf(&x).Elem()
rx.SetInt(2)                     // OK, x = 2
rx.Set(reflect.ValueOf(3))       // OK, x = 3
rx.SetString("hello")            // Аварийная ситуация:
                                 // string не присваиваемо int
rx.Set(reflect.ValueOf("hello")) // Аварийная ситуация:
                                 // string не присваиваемо int
var y interface{}
ry := reflect.ValueOf(&y).Elem()
ry.SetInt(2)                     // Аварийная ситуация:
                                 // SetInt вызван для интерфейса Value
ry.Set(reflect.ValueOf(3))       // OK, y = int(3)
ry.SetString("hello")            // Аварийная ситуация: SetString
                                 // вызван для интерфейса Value
ry.Set(reflect.ValueOf("hello")) // OK, y = "hello"
```

Применив Display к os.Stdout, мы обнаружили, что рефлексия может читать значения полей неэкспортируемой структуры, которые по обычным правилам языка

недоступны, как поле `int fd` структуры `os.File` на Unix-подобных платформах. Однако рефлексия не может обновлять такие значения:

```
stdout := reflect.ValueOf(os.Stdout).Elem() // *os.Stdout,
                                            // переменная os.File
fmt.Println(stdout.Type())                  // "os.File"
fd := stdout.FieldByName("fd")
fmt.Println(fd.Int())                       // "1"
fd.SetInt(2)                                // Аварийная ситуация:
                                            // неэкспортируемое поле
```

Адресуемое значение `reflect.Value` записывает, было ли оно получено путем обхода поля неэкспортируемой структуры, и, если это так, запрещает модификацию. Следовательно, `CanAddr` обычно не является правильной проверкой перед установкой переменной. Метод `CanSet` сообщает, является ли `reflect.Value` адресуемым *и* устанавливаемым:

```
fmt.Println(fd.CanAddr(), fd.CanSet()) // "true false"
```

12.6. Пример: декодирование S-выражений

Для каждой функции `Marshal`, предоставляемой пакетами `encoding/...` стандартной библиотеки, имеется соответствующая функция `Unmarshal`, которая выполняет декодирование. Например, как мы видели в разделе 4.5, для данного байтового среза, содержащего данные в JSON-кодировке для нашего типа `Movie` (раздел 12.3), мы можем декодировать их следующим образом:

```
data := []byte{/* ... */}
var movie Movie
err := json.Unmarshal(data, &movie)
```

Функция `Unmarshal` использует рефлексию для изменения полей существующей переменной `movie` и создания новых отображений, структур и срезов, определяемых типом `Movie` и содержимым входных данных.

Давайте теперь реализуем простую функцию `Unmarshal` для S-выражений, аналог стандартной функции `json.Unmarshal`, использованной нами ранее, и обратную к нашей функции `sexpr.Marshal`. Мы должны предупредить вас, что надежная и обобщенная реализация требует значительно большего кода, чем вы увидите в этом примере (и который и так достаточно длинный), поэтому у нас так много сокращений. Мы поддерживаем только ограниченное подмножество S-выражений и не обрабатываем ошибки. Код предназначен только для иллюстрации рефлексии, а не реального анализа S-выражения.

Лексический анализатор использует тип `Scanner` из пакета `text/scanner` для того, чтобы разделить входной поток на последовательность лексем, таких как комментарии, идентификаторы, строковые и числовые литералы. Метод `Scan` сканера выполняет проход по тексту и возвращает вид следующей лексемы, который имеет

тип rune. Большинство лексем, таких как как '(', состоят из одной руны, но пакет text/scanner представляет и многосимвольные лексемы Ident, String и Int, используя небольшие отрицательные значения типа rune. После вызова Scan, который возвращает один из этих видов лексем, метод TokenText сканера возвращает текст лексемы.

Так как типичному анализатору может понадобиться проверить текущую лексему несколько раз, а метод Scan перемещает сканер по тексту, мы "заворачиваем" сканер во вспомогательный тип lexer, который отслеживает последнюю возвращенную методом Scan лексему.

gopl.io/ch12/sexpr

```
type lexer struct {
    scan  scanner.Scanner
    token rune // the current token
}

func (lex *lexer) next()        { lex.token = lex.scan.Scan() }
func (lex *lexer) text() string { return lex.scan.TokenText() }
func (lex *lexer) consume(want rune) {
    if lex.token != want {  // Примечание: примером хорошей
                            // обработки ошибок не является!
        panic(fmt.Sprintf("получен %q, требуется %q",
                          lex.text(), want))
    }
    lex.next()
}
```

Вернемся к анализатору. Он состоит из двух основных функций. Первая из них, read, читает S-выражение, которое начинается с текущей лексемы, и обновляет переменную, на которую ссылается адресуемое значение reflect.Value v:

```
func read(lex *lexer, v reflect.Value) {
    switch lex.token {
    case scanner.Ident:
        // Единственными корректными идентификаторами
        // являются "nil" и имена полей структур.
        if lex.text() == "nil" {
            v.Set(reflect.Zero(v.Type()))
            lex.next()
            return
        }
    case scanner.String:
        s, _ := strconv.Unquote(lex.text()) // Примечание:
        v.SetString(s)                      // игнорируем ошибки
        lex.next()
        return
    case scanner.Int:
        i, _ := strconv.Atoi(lex.text())    // Примечание:
```

```
            v.SetInt(int64(i))                  // игнорируем ошибки
            lex.next()
            return
        case '(':
            lex.next()
            readList(lex, v)
            lex.next()   // Считываем ')'
            return
        }
        panic(fmt.Sprintf("неожиданная лексема %q", lex.text()))
}
```

Наши S-выражения используют идентификаторы для двух различных целей, имен полей структур и значений `nil` для указателей. Функция `read` обрабатывает только последний случай. Встретив `scanner.Ident "nil"`, она устанавливает для переменной `v` нулевое значение ее типа, используя функцию `reflect.Zero`. Для любого другого идентификатора она сообщает об ошибке. Функция `ReadList`, с которой мы вскоре познакомимся, обрабатывает идентификаторы, используемые как имена полей структуры.

Лексема `'('` указывает начало списка. Вторая функция, `readList`, декодирует список в переменную составного типа — отображение, структуру, срез или массив — в зависимости от того, какого рода переменную Go мы в настоящее время заполняем информацией. В любом случае цикл продолжает анализ элементов до тех пор, пока не встретит соответствующую закрывающую круглую скобку `')'`, как определяет функция `endList`.

Интересной частью является рекурсия. Простейший случай — массив. До встречи с закрывающей скобкой `')'` мы используем метод `Index` для получения переменной для каждого элемента массива и выполняем рекурсивный вызов `read` для его заполнения. Как и во многих других ошибочных ситуациях, если входные данные приводят к индексу за концом массива, возникает аварийная ситуация. Аналогичный подход используется для срезов, с тем отличием, что мы должны создать новую переменную для каждого элемента, заполнить ее, а затем добавить в срез.

Циклы для структур и отображений должны на каждой итерации анализировать подсписок `(key value)`. Для структур ключ является символом, определяющим поле. Как и при обработке массивов мы получаем существующую переменную для поля структуры, используя функцию `FieldByName`, и выполняем рекурсивный вызов для ее заполнения. Для отображений ключ может быть любого типа, и как и в ситуации со срезами мы создаем новую переменную, рекурсивно заполняем ее и наконец вставляем новую пару "ключ/значение" в отображение:

```
func readList(lex *lexer, v reflect.Value) {
    switch v.Kind() {
    case reflect.Array:  // (item ...)
        for i := 0; !endList(lex); i++ {
            read(lex, v.Index(i))
        }
```

```
    case reflect.Slice:  // (item ...)
        for !endList(lex) {
            item := reflect.New(v.Type().Elem()).Elem()
            read(lex, item)
            v.Set(reflect.Append(v, item))
        }
    case reflect.Struct: // ((name value) ...)
        for !endList(lex) {
            lex.consume('(')
            if lex.token != scanner.Ident {
                panic(fmt.Sprintf(
                      "получена лексема %q, требуется имя поля",
                      lex.text()))
            }
            name := lex.text()
            lex.next()
            read(lex, v.FieldByName(name))
            lex.consume(')')
        }
    case reflect.Map:    // ((key value) ...)
        v.Set(reflect.MakeMap(v.Type()))
        for !endList(lex) {
            lex.consume('(')
            key := reflect.New(v.Type().Key()).Elem()
            read(lex, key)
            value := reflect.New(v.Type().Elem()).Elem()
            read(lex, value)
            v.SetMapIndex(key, value)
            lex.consume(')')
        }
    default:
        panic(fmt.Sprintf("не могу декодировать список в %v",
                          v.Type()))
    }
}
func endList(lex *lexer) bool {
    switch lex.token {
    case scanner.EOF:
        panic("конец файла")
    case ')':
        return true
    }
    return false
}
```

Наконец мы "заворачиваем" анализатор в экспортируемую функцию `Unmarshal`, показанную ниже, что скрывает некоторые острые углы реализации. Ошибки во время анализа приводят к аварийным ситуациям, поэтому `Unmarshal` использует отло-

женный вызов для восстановления после аварийной ситуации (раздел 5.10) и возврата сообщения об ошибке.

```
// Unmarshal анализирует данные S-выражения и заполняет переменную,
// адресом которой является ненулевой указатель out.
func Unmarshal(data []byte, out interface{}) (err error) {
    lex := &lexer{scan: scanner.Scanner{Mode: scanner.GoTokens}}
    lex.scan.Init(bytes.NewReader(data))
    lex.next() // get the first token
    defer func() {
        // Примечание: это не пример идеальной обработки ошибок!
        if x := recover(); x != nil {
            err = fmt.Errorf("ошибка в %s: %v", lex.scan.Position, x)
        }
    }()
    read(lex, reflect.ValueOf(out).Elem())
    return nil
}
```

Реализация промышленного качества не должна генерировать аварийную ситуацию ни при каких входных данных, а должна просто информативно сообщать об ошибке, возможно, с номером строки или смещением в тексте. Тем не менее мы надеемся, что этот пример передает некоторое представление о том, что происходит за кулисами таких пакетов, как `encoding/json`, и как можно использовать рефлексию для заполнения структур данных.

Упражнение 12.8. Функция `sexpr.Unmarshal`, подобно `json.Unmarshal`, требует полных входных данных для байтового среза, чтобы начать декодирование. Определите тип `sexpr.Decoder`, который, подобно `json.Decoder`, позволяет декодировать последовательность значений из `io.Reader`. Измените `sexpr.Unmarshal` для использования этого нового типа.

Упражнение 12.9. Напишите API на основе лексем для декодирования S-выражений, следуя стилю `xml.Decoder` (раздел 7.14). Вам потребуются пять типов лексем: `Symbol`, `String`, `Int`, `StartList` и `EndList`.

Упражнение 12.10. Расширьте `sexpr.Unmarshal` для обработки булевых значений, чисел с плавающей точкой и интерфейсов, кодируемых вашим решением упражнения 12.3. (*Указание*: для декодирования интерфейсов потребуется отображение имени каждого поддерживаемого типа на его `reflect.Type`.)

12.7. Доступ к дескрипторам полей структур

В разделе 4.5 мы использовали *дескрипторы полей* структур для изменения JSON-кодирования значений структур Go. Дескриптор поля `json` позволяет нам выбирать альтернативные имена полей и подавлять вывод пустых полей. В этом разделе мы увидим, как обращаться к дескрипторам полей с использованием рефлексии.

Первое, что делает большинство функций-обработчиков HTTP на веб-серверах, — это извлечение параметров запроса в локальные переменные. Мы определим вспомо-

гательную функцию `params.Unpack`, которая использует дескрипторы полей структуры для более удобного написания HTTP-обработчиков (раздел 7.7).

Сначала давайте посмотрим, как она используется. Показанная ниже функция `search` является обработчиком HTTP. Она определяет переменную `data` с типом анонимной структуры, поля которой соответствуют параметрам HTTP-запроса. Теги полей структуры указывают имена параметров, которые зачастую короткие и загадочные, чтобы сэкономить драгоценное пространство в URL. Функция `Unpack` заполняет структуру информацией из запроса, так что обращение к параметрам может выполняться удобно, как к соответствующему типу данных.

gopl.io/ch12/search

```
import "gopl.io/ch12/params"

// search реализует окончание URL /search.
func search(resp http.ResponseWriter, req *http.Request) {
    var data struct {
        Labels     []string `http:"l"`
        MaxResults int      `http:"max"`
        Exact      bool     `http:"x"`
    }
    data.MaxResults = 10   // Значение по умолчанию
    if err := params.Unpack(req, &data); err != nil {
        http.Error(resp, err.Error(), http.StatusBadRequest) // 400
        return
    }
    // ... оставшаяся часть обработчика ...
    fmt.Fprintf(resp, "Поиск: %+v\n", data)
}
```

Показанная ниже функция `Unpack` делает три вещи. Сначала она вызывает `req.ParseForm()` для анализа запроса. После этого `req.Form` содержит все параметры независимо от того, что использует клиент HTTP: метод запроса `GET` или `POST`.

Затем `Unpack` создает отображение *эффективного* имени каждого поля в переменную этого поля. Эффективное имя может отличаться от фактического, если поле содержит дескриптор. Метод `Field` у `reflect.Type` возвращает значение `reflect.StructField`, предоставляющее сведения о типе каждого поля, такие как его имя, тип и необязательный дескриптор. Поле `Tag` является значением `reflect.StructTag`, которое представляет собой строковый тип, предоставляющий метод `Get` для анализа и извлечения подстроки для конкретного ключа, такой как `http:"..."` в данном случае:

gopl.io/ch12/params

```
// Unpack заполняет поля структуры, на которую указывает ptr,
// параметрами из HTTP-запроса в req.
func Unpack(req *http.Request, ptr interface{}) error {
    if err := req.ParseForm(); err != nil {
        return err
    }
```

```
    // Строит отображение с ключом, являющимся эффективным именем.
    fields := make(map[string]reflect.Value)
    v := reflect.ValueOf(ptr).Elem()   // Структурная переменная
    for i := 0; i < v.NumField(); i++ {
        fieldInfo := v.Type().Field(i) // reflect.StructField
        tag := fieldInfo.Tag           // reflect.StructTag
        name := tag.Get("http")
        if name == "" {
            name = strings.ToLower(fieldInfo.Name)
        }
        fields[name] = v.Field(i)
    }
    // Обновляет поле структуры для каждого параметра в запросе.
    for name, values := range req.Form {
        f := fields[name]
        if !f.IsValid() {
            continue // Игнорируем нераспознанные параметры HTTP
        }
        for _, value := range values {
            if f.Kind() == reflect.Slice {
                elem := reflect.New(f.Type().Elem()).Elem()
                if err := populate(elem, value); err != nil {
                    return fmt.Errorf("%s: %v", name, err)
                }
                f.Set(reflect.Append(f, elem))
            } else {
                if err := populate(f, value); err != nil {
                    return fmt.Errorf("%s: %v", name, err)
                }
            }
        }
    }
    return nil
}
```

Наконец `Unpack` перебирает пары "имя/значение" параметров HTTP и обновляет соответствующие поля структуры. Напомним, что одно и то же имя параметра может появляться более одного раза. Если это происходит и поле представляет собой срез, то все значения этого параметра накапливаются в этом срезе. В противном случае поле многократно перезаписывается, так что в результате остается записанным только последнее значение.

Функция `populate` заботится об установке одного поля `v` (или одного элемента поля, являющегося срезом) из значения параметра. В настоящее время она поддерживает только строки, знаковые целые числа и логические значения. Поддержка других типов остается читателям в качестве упражнения.

```
func populate(v reflect.Value, value string) error {
    switch v.Kind() {
```

```
    case reflect.String:
        v.SetString(value)
    case reflect.Int:
        i, err := strconv.ParseInt(value, 10, 64)
        if err != nil {
            return err
        }
        v.SetInt(i)
    case reflect.Bool:
        b, err := strconv.ParseBool(value)
        if err != nil {
            return err
        }
        v.SetBool(b)
    default:
        return fmt.Errorf("неподдерживаемый вид %s", v.Type())
    }
    return nil
}
```

Если мы добавим обработчик `server` к веб-серверу, то типичный сеанс может выглядеть следующим образом:

```
$ go build gopl.io/ch12/search
$ ./search &
$ ./fetch 'http://localhost:12345/search'
Search: {Labels:[] MaxResults:10 Exact:false}
$ ./fetch 'http://localhost:12345/search?l=golang&l=programming'
Search: {Labels:[golang programming] MaxResults:10 Exact:false}
$ ./fetch 'http://localhost:12345/search?l=golang&l=programming&max=100'
Search: {Labels:[golang programming] MaxResults:100 Exact:false}
$ ./fetch 'http://localhost:12345/search?x=true&l=golang&l=programming'
Search: {Labels:[golang programming] MaxResults:10 Exact:true}
$ ./fetch 'http://localhost:12345/search?q=hello&x=123'
x: strconv.ParseBool: parsing "123": invalid syntax
$ ./fetch 'http://localhost:12345/search?q=hello&max=lots'
max: strconv.ParseInt: parsing "lots": invalid syntax
```

Упражнение 12.11. Напишите соответствующую функцию `Pack`. Для заданного структурного значения `Pack` должна возвращать URL со значениями параметров, взятыми из структуры.

Упражнение 12.12. Расширьте запись дескриптора поля так, чтобы выражать требования корректности параметров. Например, от строки может потребоваться быть корректным адресом электронной почты или номером кредитной карты, а от целого числа — представлять собой верный почтовый индекс. Измените `Unpack` таким образом, чтобы данная функция проверяла указанные требования.

Упражнение 12.13. Измените кодировщик S-выражений из раздела 12.4 и декодировщик из раздела 12.6 так, чтобы они обрабатывали дескриптор поля `sexpr:"..."` аналогично `encoding/json` (раздел 4.5).

12.8. Вывод методов типа

Наш последний пример рефлексии использует `reflect.Type` для вывода типа произвольного значения и перечисления его методов:

gopl.io/ch12/methods

```
// Print выводит множество методов значения x.
func Print(x interface{}) {
    v := reflect.ValueOf(x)
    t := v.Type()
    fmt.Printf("type %s\n", t)

    for i := 0; i < v.NumMethod(); i++ {
        methType := v.Method(i).Type()
        fmt.Printf("func (%s) %s%s\n", t, t.Method(i).Name,
                   strings.TrimPrefix(methType.String(), "func"))
    }
}
```

И `reflect.Type`, и `reflect.Value` имеют метод под названием `Method`. Каждый вызов `t.Method(i)` возвращает экземпляр `reflect.Method`, структурный тип, который описывает имя и тип отдельного метода. Каждый вызов `v.Method(i)` возвращает значение `reflect.Value`, представляющее метод-значение (раздел 6.4), т.е. метод, связанный с его получателем. Используя метод `reflect.Value.Call` (показать который здесь нам не позволяет отсутствие места), можно вызвать `Value` разновидности `Func`, до в данной программе нас интересует только значение `Type`.

Вот методы, принадлежащие двум типам, `time.Duration` и `*strings.Replacer`:

```
methods.Print(time.Hour)
// Output:
// type time.Duration
// func (time.Duration) Hours() float64
// func (time.Duration) Minutes() float64
// func (time.Duration) Nanoseconds() int64
// func (time.Duration) Seconds() float64
// func (time.Duration) String() string

methods.Print(new(strings.Replacer))
// Output:
// type *strings.Replacer
// func (*strings.Replacer) Replace(string) string
// func (*strings.Replacer) WriteString(io.Writer, string) (int, error)
```

12.9. Предостережение

Существует гораздо больше функций API рефлексии, чем мы можем привести здесь, но показанные примеры дают представление о том, на что они способны. Рефлексия является мощным и выразительным инструментом, но по трем причинам она должна использоваться с осторожностью.

Первая причина заключается в том, что код на основе рефлексии может оказаться уязвимым. Для каждой ошибки, о которой компилятор сообщает как об ошибке типа, имеется соответствующий способ злоупотребления рефлексией; но в то время как компилятор сообщает об ошибке в ходе построения приложения, рефлексия сообщает об ошибке при выполнении программы с помощью аварийной ситуации, возможно, через длительное время после написания или даже запуска программы.

Если, например, функция `readList` (раздел 12.6) должна прочесть входную строку для заполнения переменной типа `int`, вызов `reflect.Value.SetString` приведет к аварийной ситуации. Большинство программ, которые используют рефлексию, содержат аналогичные опасности, и требуются значительные усилия для отслеживания типа, адресуемости и устанавливаемости каждого `reflect.Value`.

Наилучший способ избежать этой уязвимости — обеспечить полную инкапсуляцию применения рефлексии в вашем пакете и по возможности отказаться от `reflect.Value` в пользу конкретных типов в API вашего пакета, чтобы ограничить входные данные корректными значениями. Если это не представляется возможным, выполняйте дополнительные динамические проверки перед каждой рискованной операцией. В качестве примера из стандартной библиотеки: когда функция `fmt.Printf` применяет неподходящие для данного операнда символы преобразования, она выводит информативное сообщение об ошибке, а не ограничивается аварийным состоянием. В программе есть ошибка, но так ее куда легче диагностировать:

```
fmt.Printf("%d %s\n", "hello", 42) // "%!d(string=hello) %!s(int=42)"
```

Рефлексия также снижает безопасность и точность инструментов автоматического рефакторинга и анализа, поскольку они при этом не могут основываться на информации о типах.

Второй причиной избегать рефлексии является то, что, поскольку типы служат формой документации, а операции рефлексии не могут быть предметом статической проверки типов, активно использующий рефлексию код часто очень труден для понимания. Всегда тщательно документируйте ожидаемые типы и другие инварианты функций, которые принимают `interface{}` или `reflect.Value`.

Третья причина заключается в том, что функции с использованием рефлексии могут быть на один или два порядка медленнее, чем специализированные функции для конкретных типов. В типичной программе большинство функций не играют роли в общей производительности, так что рефлексию можно использовать там, где она делает программу более ясной. Особенно хорошо подходит рефлексия для тестирования, поскольку большинство тестов используют небольшие наборы данных. Но для функций на критическом пути выполнения рефлексии лучше избегать.

13

Низкоуровневое программирование

Дизайн Go гарантирует ряд свойств безопасности, ограничивающих пути, по которым программа Go может "пойти не туда". Во время компиляции проверка типов обнаруживает большинство попыток применить операцию к значению неподходящего типа, например попытку вычитания одной строки из другой. Строгие правила преобразования типов предотвращают прямой доступ к внутреннему содержимому встроенных типов, таких как строки, отображения, срезы и каналы.

Для ошибок, которые не могут быть обнаружены статически (таких, как обращение за пределами массива или разыменовывание нулевого указателя), динамическая проверка гарантирует, что всякий раз, когда происходят запрещенные операции, программа немедленно завершается с информативным сообщением об ошибке. Автоматическое управление памятью (сборка мусора) устраняет ошибки использования памяти после освобождения, а также большинство утечек памяти.

Многие детали реализации недоступны для программ Go. Не существует способа выяснить размещение в памяти составных типов, таких как структуры, или машинного кода функции, или потока операционной системы, в котором выполняется текущая go-подпрограмма. В действительности планировщик Go свободно перемещает go-подпрограммы из одного потока в другой. Указатель определяет переменную, не раскрывая числовое значение ее адреса. Адреса могут измениться; по мере того как сборщик мусора перемещает переменные, указатели прозрачно обновляются.

Вместе эти особенности делают программы на языке программирования Go более предсказуемыми и менее загадочными, чем программы на языке программирования C, типичном низкоуровневом языке. Сокрытие низкоуровневых деталей делает программы на Go высокопереносимыми, поскольку семантика языка не зависит от конкретного компилятора, операционной системы или архитектуры процессора. (Независимость не абсолютная: все же имеется утечка некоторых деталей, например, через размер слова процессора, порядок вычисления определенных выражений и набор ограничений реализации для конкретного компилятора.)

Но иногда мы можем предпочесть отказ от некоторых из этих полезных гарантий для достижения максимально возможной производительности, для взаимодействия с

библиотеками, написанными на других языках, или для реализации функции, которая не может быть выражена на чистом Go.

В этой главе мы увидим, как пакет `unsafe` позволяет нам выйти за пределы обычных правил и как использовать инструмент `cgo` для создания связи Go с библиотеками C и вызовами операционной системы.

Описанные в этой главе подходы не должны использоваться легкомысленно. Без пристального внимания к деталям они могут вызвать различные непредсказуемые, непостижимые и нелокальные сбои, с которыми, к несчастью, хорошо знакомы программисты на C. Использование `unsafe` также лишает Go гарантий совместимости с будущими версиями, поскольку, преднамеренно или случайно, очень легко оказаться в зависимости от неуказанных деталей реализации, которые могут неожиданно измениться.

Пакет `unsafe` довольно странный. Хотя он имеет вид обычного пакета и импортируется обычным способом, на самом деле он реализуется компилятором. Этот пакет обеспечивает доступ к ряду встроенных языковых возможностей, которые не являются общедоступными, поскольку раскрывают подробности использования памяти в Go. Представление этих функций в виде отдельного пакета делает более заметными редкие случаи, когда они необходимы. Кроме того, некоторые среды могут ограничить использование пакета `unsafe` по соображениям безопасности.

Пакет `unsafe` широко используется в таких низкоуровневых пакетах, как `runtime`, `os`, `syscall` и `net`, которые взаимодействуют с операционной системой, но он почти никогда не нужен обычным программам.

13.1. `unsafe.Sizeof`, `Alignof` и `Offsetof`

Функция `unsafe.Sizeof` сообщает размер в байтах представления ее операнда, который может быть выражением любого типа; само выражение не вычисляется. Вызов `Sizeof` является константным выражением типа `uintptr`, поэтому результат может использоваться в качестве размерности типа массива или для вычисления других констант:

```
import "unsafe"
fmt.Println(unsafe.Sizeof(float64(0))) // "8"
```

`Sizeof` сообщает размер только фиксированной части каждой структуры данных, такой как указатель или длина строки, но не косвенных частей, как, например, содержимое строки. Типичные размеры для всех не составных типов Go приведены ниже, хотя точные их размеры могут отличаться в зависимости от конкретной реализации. Для переносимости мы привели размеры ссылочных типов (типов, содержащих ссылки), выраженные в словах, где слово составляет 4 байта на 32-разрядной платформе и 8 байтов на 64-разрядной платформе.

Компьютеры наиболее эффективно загружают и сохраняют значения в памяти, когда эти значения надлежащим образом *выровнены*. Например, адрес значения двухбайтового типа, такого как `int16`, должен быть четным числом, адрес четырехбайтового

значения, такого как `rune`, должен быть кратен 4, а адрес восьмибайтового значения наподобие `float64`, `uint64` или 64-разрядного указателя должен быть кратным 8. Требования к выравниванию более высоких степеней редки, даже для больших типов данных, таких как `complex128`.

По этой причине размер значения составного типа (структуры или массива) составляет по меньшей мере сумму размеров полей или элементов, но может быть и больше из-за наличия "пробелов". Пробелы (неиспользуемая память) добавляются компилятором для обеспечения должного выравнивания следующего поля или элемента относительно начала структуры или массива

Тип	Размер
`bool`	1 байт
`intN, uintN, floatN, complex`	N/8 байтов (например, для `float64` — 8 байтов)
`int, uint, uintptr`	1 слово
`*T`	1 слово
`string`	2 слова (данные, длина)
`[]T`	3 слова (данные, длина, емкость)
`map`	1 слово
`func`	1 слово
`chan`	1 слово
`interface`	2 слова (тип, значение)

Спецификация языка не гарантирует, что порядок, в котором объявлены поля, — это порядок, в котором они располагаются в памяти, поэтому теоретически компилятор волен их переупорядочить, хотя на момент написания книги ни один компилятор так не поступает. Если типы полей структуры — различных размеров, использование памяти может оказаться эффективнее при объявлении полей в порядке, при котором они упаковываются максимально плотно. Три приведенные ниже структуры имеют одни и те же поля, но первая требует до 50% больше памяти, чем две другие:

```
                                        //  64 разряда     32 разряда
struct{ bool;    float64; int16 }       //   3 слова        4 слова
struct{ float64; int16;   bool }        //   2 слова        3 слова
struct{ bool;    int16;   float64 }     //   2 слова        3 слова
```

Детали алгоритма выравнивания выходят за рамки этой книги, и вам, безусловно, не стоит беспокоиться о каждой структуре, но эффективная упаковка может сделать часто выделяемые структуры более компактными, а потому более быстрыми.

Функция `unsafe.Alignof` сообщает о необходимом выравнивании типа ее аргумента. Как и `Sizeof`, она может быть применена к выражению любого типа и дает константу. Как правило, булевы и числовые типы выровнены по своим размерам (максимум до 8 байтов), а все другие типы выравниваются по границе слова.

Функция `unsafe.Offsetof`, операндом которой должен быть селектор поля `x.f`, вычисляет смещение поля `f` относительно начала включающей его структуры `x` с учетом всех пробелов, если таковые имеются.

На рис. 13.1 показана структурная переменная `x` и ее размещение в памяти для типичных 32- и 64-разрядных реализаций Go. Серым цветом показаны пробелы.

```
var x struct {
    a  bool
    b  int16
    c  []int
}
```

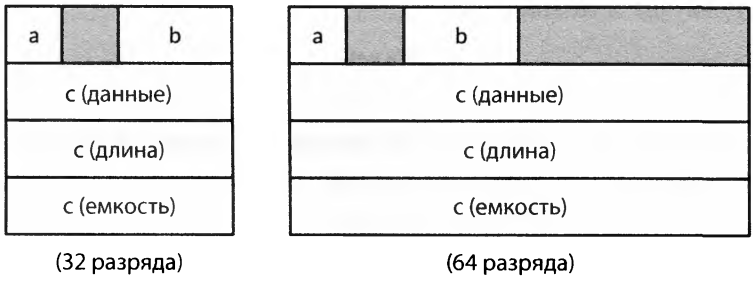


Рис. 13.1. Пробелы в структуре

В приведенной далее таблице показаны результаты применения трех рассмотренных функций из пакета `unsafe` к переменной `x` и к каждому из трех ее полей:

```
Типичная 32-разрядная платформа:
Sizeof(x)   = 16   Alignof(x)   = 4
Sizeof(x.a) =  1   Alignof(x.a) = 1   Offsetof(x.a) = 0
Sizeof(x.b) =  2   Alignof(x.b) = 2   Offsetof(x.b) = 2
Sizeof(x.c) = 12   Alignof(x.c) = 4   Offsetof(x.c) = 4

Типичная 64-разрядная платформа:
Sizeof(x)   = 32   Alignof(x)   = 8
Sizeof(x.a) =  1   Alignof(x.a) = 1   Offsetof(x.a) = 0
Sizeof(x.b) =  2   Alignof(x.b) = 2   Offsetof(x.b) = 2
Sizeof(x.c) = 24   Alignof(x.c) = 8   Offsetof(x.c) = 8
```

Несмотря на их имена, эти функции на самом деле не являются небезопасными и могут быть полезны для понимания компоновки памяти при оптимизации использования последней.

13.2. unsafe.Pointer

Большинство типов указателей записываются как `*T`, что означает "указатель на переменную типа `T`". Тип `unsafe.Pointer` представляет собой особый вид указателя, который может содержать адрес любой переменной. Конечно, мы не можем косвенно обращаться к ней через `unsafe.Pointer` с помощью выражения `*p`, так как

мы не знаем, какой тип должно иметь данное выражение. Как и обычные указатели, `unsafe.Pointer` являются сравниваемыми, и их можно сравнивать со значением `nil`, которое является нулевым значением данного типа.

Обычный указатель `*T` может быть преобразован в `unsafe.Pointer`, а `unsafe.Pointer` — обратно в обычный указатель, не обязательно того же типа, что и `*T`. Так, например, путем преобразования указателя `*float64` в `*uint64` мы можем просмотреть битовый шаблон переменной с плавающей точкой:

```
package math
func Float64bits(f float64) uint64 {
     return *(*uint64)(unsafe.Pointer(&f))
}
fmt.Printf("%#016x\n", Float64bits(1.0))  // "0x3ff0000000000000"
```

Через получающийся указатель можно также обновить битовый шаблон. Это безвредно для переменной с плавающей точкой, поскольку любой битовый шаблон является допустимым, но в общем случае преобразования `unsafe.Pointer` позволяют записывать произвольные значения в память и тем самым подрывать систему типов.

`unsafe.Pointer` можно также преобразовать в тип `uintptr`, хранящий числовое значение указателя и позволяющий выполнять арифметические операции над адресами. (Вспомните из главы 3, "Фундаментальные типы данных", что тип `uintptr` является достаточно широким, чтобы представлять адрес в виде беззнакового целого числа.) Это преобразование также может быть применено в обратном направлении, но преобразование из `uintptr` в `unsafe.Pointer` может повредить систему типов, так как не все числа представляют собой допустимые адреса.

Многие значения `unsafe.Pointer`, таким образом, являются посредниками для преобразования обычных указателей в числовые значения адресов и обратно. В приведенном ниже примере определяется адрес переменной `x` и к нему добавляется смещение ее поля `b`, полученный адрес преобразуется в указатель `*int16`, через который обновляется значение `x.b`:

gopl.io/ch13/unsafeptr

```
var x struct {
    a bool
    b int16
    c []int
}
// Эквивалентно pb := &x.b
pb := (*int16)(unsafe.Pointer(
uintptr(unsafe.Pointer(&x)) + unsafe.Offsetof(x.b)))
*pb = 42
fmt.Println(x.b) // "42"
```

Хотя данный синтаксис является громоздким (что как раз может быть неплохо, поскольку такие функции следует использовать с осторожностью), не соблазняйтесь вводить временные переменные типа `uintptr`, чтобы разбить строки. Приведенный далее код является неверным:

```
// Примечание: тонкая некорректность!
tmp := uintptr(unsafe.Pointer(&x)) + unsafe.Offsetof(x.b)
pb := (*int16)(unsafe.Pointer(tmp))
*pb = 42
```

Причина тому весьма тонкая. Некоторые сборщики мусора перемещают переменные в памяти для уменьшения фрагментации или упрощения учета. Сборщики мусора такого рода известны как *перемещающие*. При перемещении переменной все указатели, которые содержат адрес ее старого местоположения, должны быть обновлены значением ее нового адреса. С точки зрения сборщика мусора `unsafe.Pointer` является указателем и, таким образом, его значение при перемещении переменной необходимо изменить, но `uintptr` — это просто число, так что его значение изменяться не должно. Некорректный код выше *скрывает указатель* от сборщика мусора в переменной `tmp`, не являющейся указателем. Ко времени выполнения второй инструкции переменная `x` может быть перемещена, и число в `tmp` больше не будет адресом `&x.b`. В результате третья инструкция присвоит произвольной ячейке памяти значение 42.

Есть множество патологических вариаций на эту тему. После выполнения инструкции

```
pT := uintptr(unsafe.Pointer(new(T))) // Примечание: неверно!
```

нет никаких указателей, которые относятся к переменной, созданной с помощью вызова `new`, поэтому сборщик мусора имеет право ее уничтожить по окончании выполнения данной инструкции. В результате переменная `pT` будет содержать адрес, по которому переменная была, но ее больше там нет.

Ни одна из текущих реализаций Go не использует перемещающий сборщик мусора (хотя, возможно, будущие реализации будут это делать), но это не повод для самоуспокоения: текущие версии Go перемещают *некоторые* переменные в памяти. Вспомните из раздела 5.2, что при необходимости стеки go-подпрограмм могут увеличиваться. Когда это происходит, все переменные из старого стека могут быть перемещены в новый, больший стек, так что мы не можем полагаться на то, что числовое значение адреса переменной будет оставаться неизменным на протяжении всего ее жизненного цикла.

На момент написания имеется мало четких указаний о том, на что именно программисты на Go могут полагаться после преобразования `unsafe.Pointer` в `uintptr`, поэтому мы настоятельно рекомендуем обходиться минимальными предположениями. Рассматривайте все значения `uintptr` так, как будто они содержат *бывший* адрес переменной, и сводите к минимуму количество операций между преобразованием `unsafe.Pointer` в `uintptr` и использованием этого `uintptr`. В нашем первом примере, приведенном выше, все три операции — преобразование в `uintptr`, добавление смещения поля и преобразование обратно — находятся в пределах одной инструкции.

При вызове библиотечной функции, которая возвращает `uintptr`, как показанные ниже функции из пакета `reflect`, результат должен быть немедленно преобразован в `unsafe.Pointer` для гарантии, что он продолжает указывать на ту же переменную:

```
package reflect

func (Value) Pointer() uintptr
func (Value) UnsafeAddr() uintptr
func (Value) InterfaceData() [2]uintptr // (index 1)
```

13.3. Пример: глубокое равенство

Функция `DeepEqual` из пакета `reflect` сообщает, являются ли два значения "глубоко" равными. `DeepEqual` сравнивает значения фундаментальных типов, как если бы использовался встроенный оператор ==; для составных значений она обходит их рекурсивно, сравнивая соответствующие элементы. Работая с любой парой значений, даже с теми, которые не сравниваемы с помощью ==, она находит широкое применение в тестах. Следующий тест использует `DeepEqual` для сравнения двух значений `[]string`:

```
func TestSplit(t *testing.T) {
    got := strings.Split("a:b:c", ":")
    want := []string{"a", "b", "c"};
    if !reflect.DeepEqual(got, want) { /* ... */ }
}
```

Хотя `DeepEqual` удобна, ее сравнение может показаться достаточно произвольным. Например, она не рассматривает нулевое отображение равным ненулевому пустому отображению, как и не рассматривает нулевой срез равным ненулевому пустому срезу:

```
var a, b []string = nil, []string{}
fmt.Println(reflect.DeepEqual(a, b))     // "false"
var c, d map[string]int = nil, make(map[string]int)
fmt.Println(reflect.DeepEqual(c, d))     // "false"
```

В этом разделе мы определим функцию `Equal`, которая сравнивает произвольные значения. Подобно `DeepEqual`, она сравнивает срезы и отображения на основе их элементов, но в отличие от `DeepEqual` считает нулевой срез (или отображение) равным ненулевому пустому срезу (или отображению). Основная рекурсия над аргументами может быть выполнена с помощью рефлексии, с использованием подхода, аналогичного подходу программы `Display`, которую мы видели в разделе 12.3. Как обычно, для рекурсии мы определяем неэкспортируемую функцию `equal`. Пока что не беспокойтесь о параметре `seen`. Для каждой пары сравниваемых значений `x` и `y` функция `equal` проверяет, что они оба (или ни одно из них) являются действительными и что имеют один и тот же тип. Результат функции определяется как множество сравнений двух значений одного типа. По соображениям экономии бумаги мы опустили несколько инструкций `case`, поскольку данный шаблон должен быть вам хорошо знаком:

gopl.io/ch13/equal

```
func equal(x, y reflect.Value, seen map[comparison]bool) bool {
    if !x.IsValid() || !y.IsValid() {
        return x.IsValid() == y.IsValid()
    }
    if x.Type() != y.Type() {
        return false
    }
    // ... проверка циклов опущена (будет показана позже)...
    switch x.Kind() {
    case reflect.Bool:
        return x.Bool() == y.Bool()
    case reflect.String:
        return x.String() == y.String()
    // ... числовые типы опущены для краткости ...
    case reflect.Chan, reflect.UnsafePointer, reflect.Func:
        return x.Pointer() == y.Pointer()
    case reflect.Ptr, reflect.Interface:
        return equal(x.Elem(), y.Elem(), seen)
    case reflect.Array, reflect.Slice:
        if x.Len() != y.Len() {
            return false
        }
        for i := 0; i < x.Len(); i++ {
            if !equal(x.Index(i), y.Index(i), seen) {
                return false
            }
        }
        return true
        // ... структуры и отображения опущены для краткости ...
    }
    panic("недоступен")
}
```

Как обычно, мы не показываем использование рефлексии в API, поэтому экспортируемая функция `Equal` должна вызывать `reflect.ValueOf` для своих аргументов:

```
// Equal сообщает о глубоком равенстве x и y.
func Equal(x, y interface{}) bool {
    seen := make(map[comparison]bool)
    return equal(reflect.ValueOf(x), reflect.ValueOf(y), seen)
}

type comparison struct {
    x, y   unsafe.Pointer
    t      reflect.Type
}
```

Чтобы гарантировать, что алгоритм завершается даже для циклических структур, он должен записывать, какие пары переменных уже сравнивались, и избегать их повторного сравнения. `Equal` выделяет память для множества структур `comparison`, каждая из которых хранит адреса двух переменных (представленных значениями `unsafe.Pointer`) и тип сравнения. Помимо адреса, нужно записывать тип, потому что различные переменные могут иметь один и тот же адрес. Например, если `x` и `y` оба являются массивами, то `x` и `x[0]` имеют один и тот же адрес, как и `y` и `y[0]`, и нам важно отличить, сравниваем ли мы `x` и `y` или `x[0]` и `y[0]`.

После того как функция `equal` установила, что аргументы имеют одинаковый тип, и прежде чем она выполнит инструкцию `switch`, она проверяет, не были ли уже рассмотрены эти переменные, и если это так, прекращает рекурсию.

```
// Проверка цикла
if x.CanAddr() && y.CanAddr() {
    xptr := unsafe.Pointer(x.UnsafeAddr())
    yptr := unsafe.Pointer(y.UnsafeAddr())
    if xptr == yptr {
        return true // Идентичные ссылки
    }
    c := comparison{xptr, yptr, x.Type()}
    if seen[c] {
        return true // Уже просмотрено
    }
    seen[c] = true
}
```

Вот как выглядит наша функция `Equal` в действии:

```
fmt.Println(Equal([]int{1, 2, 3}, []int{1, 2, 3}))        // "true"
fmt.Println(Equal([]string{"foo"}, []string{"bar"}))      // "false"
fmt.Println(Equal([]string(nil), []string{}))             // "true"
fmt.Println(Equal(map[string]int(nil), map[string]int{})) // "true"
```

Она работает даже для циклических входных данных, похожих на те, которые заставляли зациклиться функцию `Display` из раздела 12.3:

```
// Циклические связанные списки a -> b -> a и c -> c.
type link struct {
    value  string
    tail   *link
}
a, b, c := &link{value: "a"}, &link{value: "b"}, &link{value: "c"}
a.tail, b.tail, c.tail = b, a, c
fmt.Println(Equal(a, a))    // "true"
fmt.Println(Equal(b, b))    // "true"
fmt.Println(Equal(c, c))    // "true"
fmt.Println(Equal(a, b))    // "false"
fmt.Println(Equal(a, c))    // "false"
```

Упражнение 13.1. Определите функцию глубокого сравнения, которая рассматривает числа (любого типа) равными, если они различаются меньше чем на одну миллиардную.

Упражнение 13.2. Напишите функцию, которая сообщает, является ли аргумент циклической структурой данных.

13.4. Вызов кода C с помощью cgo

Программе на языке Go может потребоваться использовать аппаратный драйвер, реализованный на C, встроенную базу данных, реализованную на C++, или некоторые подпрограммы линейной алгебры, написанные на Fortran. Язык программирования C уже давно является чем-то вроде "языка межнационального общения" в программировании, так что многие пакеты, предназначенные для широкого использования, экспортируют C-совместимый API, независимо от языка их реализации.

В этом разделе мы создадим простую программу для сжатия данных, которая использует `cgo`, инструмент, который связывает Go с функциями на языке C. Такие инструменты называются *интерфейсы с функциями на других языках* (foreign-function interfaces — FFI), и `cgo` является не единственным для программ Go. Еще одним является SWIG (`swig.org`); он обеспечивает более сложные возможности интеграции с классами C++, но мы не будем рассматривать его здесь.

Поддерево `compress/...` стандартной библиотеки обеспечивает сжатие и распаковку с помощью популярных алгоритмов сжатия, включая LZW (используется командой `compress` Unix) и DEFLATE (используется командой GNU `gzip`). API-интерфейсы этих пакетов незначительно различаются в деталях, но все они предоставляют оболочку для интерфейса `io.Writer`, которая сжимает данные, записываемые в него, и программу-оболочку для интерфейса `io.Reader`, которая распаковывает считываемые из него данные, например:

```
package gzip     // compress/gzip
func NewWriter(w io.Writer) io.WriteCloser
func NewReader(r io.Reader) (io.ReadCloser, error)
```

Алгоритм bzip2, основанный на элегантном преобразовании Барроуза–Уилера (Burrows-Wheeler transform), работает медленнее, чем gzip, но дает значительно лучшее сжатие. Пакет `compress/bzip2` предоставляет распаковщик для bzip2, но в данный момент пакет не предоставляет упаковщик. Создание его "с нуля" является огромной задачей, но есть хорошо документированная, высокопроизводительная реализация с открытым исходным кодом на C, пакет `libbzip2`, доступный по адресу `bzip.org`.

Если библиотека C небольшая, ее можно просто перенести на чистый Go, а если производительность не имеет решающего значения для наших целей, можно просто вызывать программу на C как вспомогательный подпроцесс с использованием пакета `os/exec`. Но когда нужно использовать сложную библиотеку, критичную в плане производительности, и с узким API на языке C, может иметь смысл использовать ее с

помощью cgo. В остальной части этой главы мы рассмотрим пример такого использования.

Из пакета libbzip2 на С нам нужен структурный тип bz_stream, который хранит входной и выходной буфера, и три функции на С: BZ2_bzCompressInit, которая выделяет буфера потоков; BZ2_bzCompress, которая сжимает данные из входного буфера в выходной; и BZ2_bzCompressEnd, которая освобождает буфера. (Не беспокойтесь о механизме пакета libbzip2; цель данного примера — просто показать, как заставить все это работать вместе.)

Мы будем вызывать функции С BZ2_bzCompressInit и BZ2_bzCompressEnd непосредственно из Go, но для BZ2_bzCompress мы определим функцию-оболочку на С, чтобы показать, как это делается. Представленный ниже исходный файл на С находится вместе с кодом Go в нашем пакете:

gopl.io/ch13/bzip

```
/* Это файл gopl.io/ch13/bzip/bzip2.c, простая */
/* оболочка для libbzip2, пригодная для cgo.   */
#include <bzlib.h>

int bz2compress(bz_stream *s, int action,
                char *in, unsigned *inlen, char *out,
                unsigned *outlen) {
    s->next_in   = in;
    s->avail_in  = *inlen;
    s->next_out  = out;
    s->avail_out = *outlen;
    int r = BZ2_bzCompress(s, action);
    *inlen  -= s->avail_in;
    *outlen -= s->avail_out;
    return r;
}
```

Теперь давайте обратимся к коду Go, первая часть которого показана ниже. Объявление import "C" является особенным. Не существует пакета "C", но это объявление заставляет go build выполнить предварительную обработку файла с помощью инструмента cgo перед тем, как он попадет компилятору Go.

```
// Пакет bzip предоставляет writer, который
// использует сжатие bzip2 (bzip.org).
package bzip
/*
#cgo CFLAGS:  -I/usr/include
#cgo LDFLAGS: -L/usr/lib -lbz2
#include <bzlib.h>
int bz2compress(bz_stream *s, int action,
                char *in, unsigned *inlen, char *out,
                unsigned *outlen);
*/
```

```
import "C"
import (
    "io"
    "unsafe"
)

type writer struct {
    w      io.Writer // Выходной поток
    stream *C.bz_stream
    outbuf [64 * 1024]byte
}

// NewWriter возвращает writer для сжатых потоков.
func NewWriter(out io.Writer) io.WriteCloser {
    const (
        blockSize  =  9
        verbosity  =  0
        workFactor = 30
    )
    w := &writer{w: out, stream: new(C.bz_stream)}
    C.BZ2_bzCompressInit(w.stream, blockSize, verbosity, workFactor)
    return w
}
```

Во время предварительной обработки `cgo` создает временный пакет, содержащий объявления Go, соответствующие всем C-функциям и типам, используемым файлом, таким как `C.bz_stream` и `C.BZ2_bzCompressInit`. Инструмент `cgo` обнаруживает эти типы путем специального вызова компилятора C для содержимого комментария, который предшествует объявлению импорта.

Комментарий может содержать директивы `#cgo`, определяющие дополнительные опции для инструментов C. Значения `CFLAGS` и `LDFLAGS` указывают дополнительные аргументы для команд компилятора и компоновщика, чтобы они могли найти заголовочный файл `bzlib.h` и библиотеку `libbz2.a`. В примере предполагается, что они установлены в каталоге `/usr` в вашей системе. Вам может потребоваться изменить или удалить эти флаги для своей конкретной установки.

`NewWriter` выполняет вызов функции C `BZ2_bzCompressInit` для инициализации буферов для потоков. Тип `writer` включает в себя еще один буфер, который будет использоваться для опустошения выходного буфера декомпрессора.

Метод `Write`, показанный ниже, передает несжатые данные `data` компрессору, в цикле вызывает функцию `bz2compress`, пока все данные не будут обработаны. Обратите внимание, что программа Go может обращаться к типам C, таким как `bz_stream`, `char` и `uint`, к функциям наподобие `bz2compress` и даже к макросам препроцессора, таким как `BZ_RUN`; везде используется запись `C.x`. Тип `C.uint` отличается от типа `uint` Go, даже если оба имеют одинаковую ширину:

```
func (w *writer) Write(data []byte) (int, error) {
    if w.stream == nil {
```

```
        panic("закрыт")
    }
    var total int    // Записанных несжатых байтов
    for len(data) > 0 {
        inlen, outlen := C.uint(len(data)), C.uint(cap(w.outbuf))
        C.bz2compress(w.stream, C.BZ_RUN,
                (*C.char)(unsafe.Pointer(&data[0])), &inlen,
                (*C.char)(unsafe.Pointer(&w.outbuf)), &outlen)
        total += int(inlen)
        data = data[inlen:]
        if _, err := w.w.Write(w.outbuf[:outlen]); err != nil {
            return total, err
        }
    }
    return total, nil
}
```

Каждая итерация цикла передает функции `bz2compress` адрес и длину оставшейся части данных `data`, а также адрес и емкость `w.outbuf`. Две переменные длины передаются по их адресам, а не по значениям, так что функция C может обновить их, чтобы указать, каково количество потребленных несжатых данных и каков размер сжатых данных. Затем каждый блок сжатых данных записывается в базовый `io.Writer`.

Метод `Close` имеет структуру, аналогичную `Write`, используя цикл для сброса всех оставшихся сжатых данных из выходного буфера потока:

```
// Close сбрасывает сжатые данные и закрывает поток.
// Он не закрывает базовый io.Writer.
func (w *writer) Close() error {
    if w.stream == nil {
        panic("закрыт")
    }
    defer func() {
        C.BZ2_bzCompressEnd(w.stream)
        w.stream = nil
    }()
    for {
        inlen, outlen := C.uint(0), C.uint(cap(w.outbuf))
        r := C.bz2compress(w.stream, C.BZ_FINISH, nil, &inlen,
                (*C.char)(unsafe.Pointer(&w.outbuf)), &outlen)
        if _, err := w.w.Write(w.outbuf[:outlen]); err != nil {
            return err
        }
        if r == C.BZ_STREAM_END {
            return nil
        }
    }
}
```

После завершения `Close` вызывает `C.BZ2_bzCompressEnd` для освобождения буферов потока, используя `defer`, чтобы гарантировать, что это произойдет на всех путях выхода. В этот момент указатель `w.stream` больше не является безопасным для разыменования. Для защиты мы приравниваем его к `nil` и добавляем явную проверку на `nil` в каждый метод, так что, если пользователь по ошибке вызывает метод после `Close`, возникает аварийная ситуация.

Не только `writer` не является безопасным с точки зрения параллельности, но и одновременные вызовы `Close` и `Write` могут привести к аварийному завершению кода C. Исправить ситуацию предлагается в упражнении 13.3.

Приведенная ниже программа `bzipper` является командой упаковки `bzip2`, которая использует наш новый пакет. Она ведет себя как команда `bzip2`, имеющаяся во многих системах Unix:

gopl.io/ch13/bzipper

```
// Bzipper читает входные данные, сжимает их с помощью
// bzip2compresses и записывает в стандартный вывод.
package main

import (
    "io"
    "log"
    "os"
    "gopl.io/ch13/bzip"
)

func main() {
    w := bzip.NewWriter(os.Stdout)
    if _, err := io.Copy(w, os.Stdin); err != nil {
        log.Fatalf("bzipper: %v\n", err)
    }
    if err := w.Close(); err != nil {
        log.Fatalf("bzipper: закрыт: %v\n", err)
    }
}
```

В приведенном ниже сеансе работы мы используем `bzipper` для сжатия файла `/usr/share/dict/words`, системного словаря, с 938 848 байтов до 335 405 байтов (около трети исходного размера), а затем распаковываем его с помощью системной команды `bunzip2`. Хеш SHA256 оказывается одним и тем же до упаковки и после распаковки, что дает нам уверенность в правильной работе программы сжатия. (Если в вашей системе нет `sha256sum`, воспользуйтесь своим решением упражнения 4.2).

```
$ go build gopl.io/ch13/bzipper
$ wc -c < /usr/share/dict/words
938848
$ sha256sum < /usr/share/dict/words
126a4ef38493313edc50b86f90dfdaf7c59ec6c948451eac228f2f3a8ab1a6ed -
$ ./bzipper < /usr/share/dict/words | wc -c
```

```
335405
$ ./bzipper < /usr/share/dict/words | bunzip2 | sha256sum
126a4ef38493313edc50b86f90dfdaf7c59ec6c948451eac228f2f3a8ab1a6ed -
```

Мы продемонстрировали компоновку библиотеки C в программу Go. Можно работать и в обратном направлении, компилируя программу на Go как статический архив, который может быть связан с программой на C, или как динамически загружаемую библиотеку, которая может быть использована программой на C. Мы только бегло взглянули на `cgo`; на самом деле есть куда больше важных вопросов, таких как управление памятью, указатели, обратные вызовы, обработка сигналов, строки, `errno`, финализаторы и отношения между go-подпрограммами и потоками операционной системы, по большей части очень тонких... В частности, правила корректной передачи указателей из Go в C и обратно оказываются очень сложными по причинам, аналогичным обсуждавшимся в разделе 13.2, и до сих пор окончательно не согласованными. В качестве дальнейшего чтения обратитесь к материалам по адресу `https://golang.org/cmd/cgo`.

Упражнение 13.3. Воспользуйтесь `sync.Mutex`, чтобы сделать `bzip2.writer` безопасным с точки зрения параллельного использования многими go-подпрограммами.

Упражнение 13.4. Зависимость от библиотек C имеет свои недостатки. Создайте альтернативную реализацию `bzip.NewWriter` на чистом Go, которая использует пакет `os/exec` для выполнения `/bin/bzip2` в качестве подпроцесса.

13.5. Еще одно предостережение

Мы закончили предыдущую главу предупреждением о недостатках интерфейса рефлексии. Это предупреждение имеет еще бо́льшую силу в случае пакета `unsafe`, описанного в этой главе.

Высокоуровневые языки изолируют программы и программистов не только от специфики наборов команд отдельных компьютеров, но и от зависимости от таких подробностей, как размещение переменных в памяти, размеры типов данных и целого ряда иных деталей реализации. Этот изолирующий слой позволяет писать безопасные и надежные программы, которые без изменений работают в любой операционной системе.

Пакет `unsafe` позволяет программистам преодолеть эту изоляцию для использования некоторых важных, но недоступных иным путем функций или, возможно, для достижения более высокой производительности. Обычно платой за это являются переносимость и безопасность, поэтому вы используете пакет `unsafe` на свой страх и риск. Наши советы о том, как и когда использовать пакет `unsafe`, напоминают рекомендации Кнута (Knuth) о преждевременной оптимизации, которые мы цитировали в разделе 11.5. Большинству программистов вообще никогда не придется использовать пакет `unsafe`. Тем не менее иногда могут возникать ситуации, когда некоторый критический фрагмент кода может быть написан более эффективно с использованием `unsafe`. Если тщательное изучение и измерения указывают, что применение `unsafe` действительно является наилучшим выходом, ограничьте его применение

как можно меньшей областью, чтобы бо́льшая часть программы ничего не знала о его применении.

А теперь спрячьте материал последних двух глав в загашник своей памяти. Вернитесь к ним не ранее чем напишете далеко не одну нетривиальную программу на языке программирования Go. Но и тогда избегайте `reflect` и `unsafe` настолько, насколько это будет возможно.

Счастливого Go-программирования! Мы надеемся, что оно понравится вам ничуть не меньше, чем нам.

Предметный указатель

Т

У

Ф

Ц

Ч

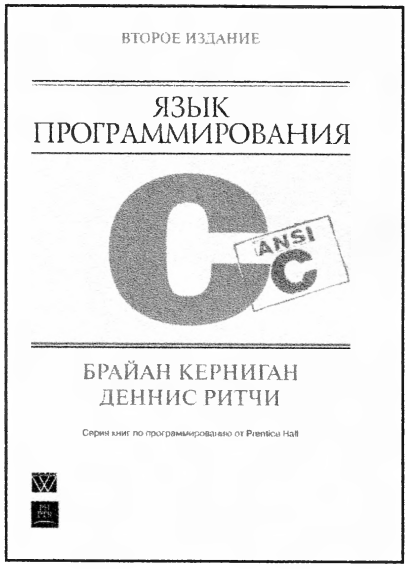
Ш

Э

Я

ЯЗЫК ПРОГРАММИРОВАНИЯ С
2-е издание

Брайан Керниган,
Деннис Ритчи



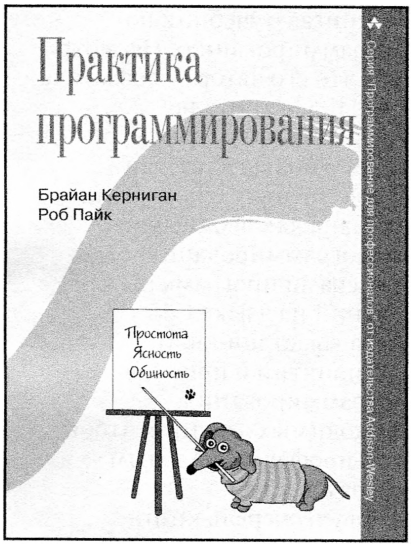
www.williamspublishing.com

Классическая книга по языку С, написанная самими разработчиками этого языка. Книга является как практически исчерпывающим справочником, так и учебным пособием по самому распространенному языку программирования. Предлагаемое второе издание книги было существенно переработано по сравнению с первым в связи с появлением стандарта ANSI C, для которого она частично послужила основой. Для изучения книги требуются знания основ программирования и вычислительной техники. Книга предназначена для широкого круга программистов и компьютерных специалистов. Может использоваться как учебное пособие для вузов.

ISBN 978-5-8459-1975-5 **в продаже**

ПРАКТИКА ПРОГРАММИРОВАНИЯ

Б. Керниган
Р. Пайк



www.williamspublishing.com

Брайану Кернигану и Робу Пайку удалось придать своей книге *Практика программирования* ту же глубину и профессиональное качество, которые характеризуют их другую классическую совместную работу *The Unix Programming Environment*. Эта книга поможет любому программисту сделать свой труд более производительным и эффективным.
Практика программирования состоит не только из написания кода. Программисты должны также оценивать затраты и приобретения, выбирать между архитектурными альтернативами, отлаживать и тестировать код, оптимизировать быстродействие, дорабатывать и обслуживать программы, написанные ими же или их коллегами. Одновременно необходимо заботиться о совместимости, стабильности и надежности программ, при этом удовлетворяя стандартам и спецификациям. Авторы вложили в эту книгу концентрированный опыт многих лет программирования, преподавания и совместной работы с коллегами. Всякий, кому приходится писать программы, почерпнет немало полезного из принципов и рекомендаций, приведенных в книге.

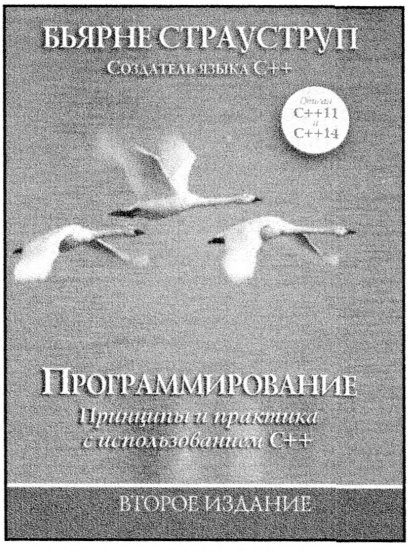
ISBN 978-5-8459-2005-8

в продаже

ПРОГРАММИРОВАНИЕ.
ПРИНЦИПЫ И ПРАКТИКА С ИСПОЛЬЗОВАНИЕМ C++

ВТОРОЕ ИЗДАНИЕ

Бьярне Страуструп



www.williamspublishing.com

Эта книга — учебник по программированию. Несмотря на то что его автор — создатель языка C++, книга не посвящена этому языку; он играет в большей степени иллюстративную роль. Книга задумана как вводный курс по программированию с примерами программных решений на языке C++ и описывает широкий круг понятий и приемов программирования, необходимых для того, чтобы стать профессиональным программистом.

В первую очередь книга адресована начинающим программистам, но она будет полезна и профессионалам, которые найдут в ней много новой информации, а главное, смогут узнать точку зрения создателя языка C++ на современные методы программирования.

ISBN 978-5-8459-1949-6 **в продаже**

Программирование/язык Go

Язык программирования Go

Данная книга является важным и авторитетным источником знаний для тех, кто хочет изучить язык программирования Go. В ней идет речь о том, как писать ясные и идиоматические программы на языке Go для решения реальных практических задач. Книга не предполагает ни наличия некоторых предварительных знаний языка Go, ни опыта работы с каким-то конкретным языком программирования, так что она будет доступна для вас, с каким бы языком вы раньше ни работали — JavaScript, Ruby, Python, Java или C++.

- Первая глава представляет собой справочник основных концепций языка Go, показанных на примере программ файлового ввода-вывода и обработки текстов, простейшей графики, а также веб-клиентов и серверов.
- Первые главы охватывают структурные элементы программ на языке Go: синтаксис, управление потоком выполнения, типы данных и организация программ в виде пакетов, файлов и функций. Приводимые примеры иллюстрируют множество пакетов стандартной библиотеки и показывают, как создавать собственные пакеты. В последующих главах механизм пакетов будет рассмотрен более подробно, и вы узнаете, как строить, тестировать и поддерживать проекты, используя инструментарий `go`.
- В главах, посвященных методам и интерфейсам, описывается нестандартный подход языка Go к объектно-ориентированному программированию, в котором методы могут быть объявлены для любого типа, а интерфейсы — неявно удовлетворены. В этих главах поясняются ключевые принципы инкапсуляции, композиции и заменимости с использованием реалистичных примеров.
- Две главы, посвященные параллелизму, представляют углубленный подход к этой важной теме. Первая глава охватывает основные механизмы go-подпрограмм и каналов, иллюстрирует стиль, известный как взаимодействие последовательных процессов, которым знаменит Go. Вторая охватывает более традиционные аспекты параллелизма с совместно используемыми переменными. Эти главы послужат прочным фундаментом для программистов, которые впервые встречаются с параллельными вычислениями.
- В двух последних главах рассматриваются низкоуровневые возможности языка Go. Одна из них посвящена искусству метапрограммирования с помощью рефлексии, а другая показывает, как использовать пакет `unsafe` для выхода за пределы системы типов в особых случаях, а также как использовать инструмент `cgo` для связи Go с библиотеками C.

В книге приведены сотни интересных и практичных примеров хорошо написанного кода на языке Go, которые охватывают весь язык, его наиболее важные пакеты и имеют широкий спектр применения. В каждой главе содержатся упражнения для проверки вашего понимания и изучения возможных расширений и альтернатив. Исходные тексты свободно доступны для загрузки по адресу `http://gopl.io/` и могут быть легко получены, построены и инсталлированы с использованием команды `go get`.

Алан А. А. Донован является членом команды разработчиков языка Go в Google (Нью-Йорк). Он получил ученую степень в области информатики в Кембридже и в МТИ занимается программированием с 1996 года. Начиная с 2005 года, он работает в Google над проектами в области инфраструктуры и был одним из разработчиков патентованной системы построения Blaze. Им создано множество библиотек и инструментов для статического анализа программ на языке Go, включая `oracle`, `godoc-analysis`, `eg` и `gorename`.

Брайан У. Керниган — профессор факультета информатики в Принстонском университете. С 1969 по 2000 год он работал в исследовательском центре в Bell Labs, где занимался языками и инструментами для Unix. Он является одним из авторов ряда книг, включая такие, как *Язык программирования C, 2-е издание* (пер. с англ., ИД "Вильямс", 2006) и *Практика программирования* (пер. с англ., ИД "Вильямс", 2005).

informit.com/series/professionalcomputing
informit.com/aw
gopl.io

Издательский дом "Вильямс"
www.williamspublishing.com

Addison-Wesley

ISBN 978-5-8459-2051-5

16010

9 785845 920515

ALWAYS LEARNING

PEARSON