Программирование на языке Go

Марк Саммерфильд

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Марк Саммерфильд

Программирование на Go

Разработка приложений XXI века

Mark Summerfield

Programming in Go:

Creating Applications for the 21st Century

Addison-Wesley

Марк Саммерфильд

Программирование на Go

Разработка приложений XXI века

2-е издание, электронное

ДМК Пресс
издательство

Москва, 2023

УДК 004.438Go
ББК 32.973.26-018.1
С17

Саммерфильд, Марк.
С17 Программирование на Go. Разработка приложений XXI века / М. Саммерфильд ; пер. с англ. А. Н. Киселёва. — 2-е изд., эл. — 1 файл pdf : 581 с. — Москва : ДМК Пресс, 2023. — Систем. требования: Adobe Reader XI либо Adobe Digital Editions 4.5 ; экран 10». — Текст : электронный.

ISBN 978-5-89818-611-1

На сегодняшний день Go — самый впечатляющий из новых языков программирования. Изначально он создавался для того, чтобы помочь задействовать всю мощь современных многоядерных процессоров. В этом руководстве Марк Саммерфильд, один из основоположников программирования на языке Go, показывает, как писать программы, в полной мере использующие его революционные возможности и идиомы.

Данная книга представляет собой одновременно и учебник, и справочник, сводя воедино все знания, необходимые для того, чтобы продолжать освоение Go, думать на Go и писать на нем высокопроизводительные программы. Автор приводит множество сравнений идиом программирования, демонстрируя преимущества Go перед более старыми языками и уделяя особое внимание ключевым инновациям. Попутно, начиная с самых основ, Марк Саммерфильд разъясняет все аспекты параллельного программирования на языке Go с применением каналов и без использования блокировок, а также показывает гибкость и необычность подхода к объектно-ориентированному программированию с применением механизма динамической типизации.

Издание предназначено для программистов разной квалификации, желающих освоить и применять в своей практике язык Go.

УДК 004.438Go
ББК 32.973.26-018.1

Электронное издание на основе печатного издания: Программирование на Go. Разработка приложений XXI века / М. Саммерфильд ; пер. с англ. А. Н. Киселёва. — Москва : ДМК Пресс, 2016. — 580 с. — ISBN 978-5-97060-338-3. — Текст : непосредственный.

ISBN 978-5-89818-611-1

Содержание

Эта книга посвящается
Жасмин Бланшетт (Jasmin Blanchette) и Трентону Шульцу (Trenton Schulz)

Введение

Цель этой книги – научить специфике программирования на языке Go с использованием всех его характерных особенностей, а также рассказать о наиболее часто применяемых пакетах, входящих в состав стандартной библиотеки Go. Книга также задумывалась как справочник для тех, кто уже знаком с языком. Чтобы соответствовать этим двум целям, была сделана попытка охватить сразу все темы в одной книге, и в текст были добавлены перекрестные ссылки.

По духу Go подобен языку C – это компактный и эффективный язык программирования с низкоуровневыми возможностями, такими как указатели. Однако Go обладает множеством особенностей, характерных для высоко- и очень высокоуровневых языков, таких как поддержка строк Юникода, высокоуровневые структуры данных, динамическая типизация, автоматическая сборка мусора и высокоуровневая поддержка взаимодействий, основанных на обмене сообщениями, а не на блокировках и разделяемых данных. Кроме того, язык Go имеет обширную и всестороннюю стандартную библиотеку.

Предполагается, что читатель уже имеет опыт программирования на распространенных языках, таких как C, C++, Java, Python и им подобных, тем не менее все уникальные особенности и идиомы языка Go демонстрируются на законченных, работающих примерах, подробно описываемых в тексте.

Для успешного освоения любого языка программирования совершенно необходимо писать программы на этом языке. С этой точки зрения в книге предпринят абсолютно практический подход: читателям предлагается не бояться экспериментировать с примерами, выполнять предлагаемые упражнения и писать собственные программы, чтобы обрести практический опыт. Как и во всех моих предыдущих книгах, все фрагменты программного кода являются «живым кодом», то есть этот код автоматически извлекался из исходных файлов `.go` и вставлялся в документ PDF перед передачей издателю, поэтому все примеры гарантированно работоспособны и

в них исключены ошибки, возможные при копировании вручную. Везде, где только возможно, в качестве примеров демонстрируются небольшие, но законченные программы и пакеты. Все примеры, упражнения и решения доступны на сайте www.qtrac.eu/gobook.html.

Основной целью книги является обучение *языку* Go, здесь рассказывается о многих пакетах из стандартной библиотеки Go, но далеко не обо всех. Однако в этом нет никакой проблемы, поскольку книга дает достаточный объем информации о Go, чтобы читатель смог самостоятельно использовать любые стандартные или сторонние пакеты и конечно же создавать собственные.

Зачем изучать язык Go?

Разработка языка Go началась в 2007 году как внутренний проект компании Google. Оригинальная архитектура языка была разработана Робертом Гризмером (Robert Griesemer) и корифеями ОС Unix – Робом Пайком (Rob Pike) и Кеном Томпсоном (Ken Thompson). 10 ноября 2009 года были опубликованы исходные тексты реализации языка Go под либеральной открытой лицензией. Развитие языка Go продолжается группой разработчиков из компании Google, в состав которой входят основатели языка, а также Расс Кокс (Russ Cox), Эндрю Джерранд (Andrew Gerrand), Ян Ланс Тейлор (Ian Lance Taylor) и многие другие. Разработка ведется с использованием открытой модели, благодаря чему в процессе участвуют многие разработчики со всего мира, порой настолько известные и уважаемые, что им предоставлены те же привилегии доступа к репозиторию с исходными текстами, что и специалистам из компании Google. Кроме того, в общественном доступе имеется множество сторонних пакетов для языка Go, доступных на сайте Go Dashboard (godashboard.appspot.com/project).

Go – один из самых удивительных языков, появившихся в последние 15 лет, и первый, нацеленный на программистов и компьютеры XXI века.

Go проектировался с прицелом на эффективное масштабирование, благодаря чему его можно использовать для создания очень больших приложений и компиляции даже очень больших программ за секунды на единственном компьютере. Молниеносная скорость компиляции обеспечивается отчасти простотой синтаксического анализа программ на этом языке, но главным образом благодаря особенностям управления зависимостями. Например, если файл `app.`

go зависит от файла pkg1.go, который, в свою очередь, зависит от файла pkg2.go, в обычных компилирующих языках для компиляции файла app.go необходимо иметь объектные модули, полученные в результате компиляции обоих файлов, pkg1.go и pkg2.go. Но в Go все, что экспортирует pkg2.go, включено в объектный модуль для файла pkg1.go, поэтому для компиляции app.go достаточно иметь только объектный модуль для файла pkg1.go. Для случая с тремя файлами это едва ли имеет большое значение, но в огромных приложениях с большим количеством зависимостей эта особенность дает весьма значительный прирост скорости компиляции.

Благодаря высокой скорости компиляции программ на языке Go появляется возможность использовать этот язык в областях, где обычно применяются языки сценариев (см. врезку «Сценарии на языке Go» ниже). Кроме того, язык Go можно использовать для создания веб-приложений с применением Google App Engine.

Язык Go имеет очень простой и понятный синтаксис, в котором отсутствуют сложные и замысловатые конструкции, характерные для более старых языков, таких как C++ (появившегося в 1983 году) или Java (появившегося в 1995 году). И относится к категории языков со строгой статической типизацией, что многими программистами считается важным условием для разработки крупных программ. Однако система типов данных в языке Go не слишком обременительна благодаря поддержке синтаксиса объявления переменных одновременно с их инициализацией (когда компилятор определяет тип автоматически, избавляя от необходимости явно указывать его) и наличию мощного и удобного механизма динамической типизации.

Языки программирования, такие как C и C++, требуют от программистов выполнения массы работы, когда дело доходит до управления памятью, которую можно переложить на плечи компьютера, особенно в многопоточных приложениях, где учет использования динамической памяти может оказаться невероятно сложной задачей. В последние годы ситуация в этой области в языке C++ намного улучшилась благодаря появлению «интеллектуальных» указателей, но он пока не способен догнать язык Java с его библиотекой поддержки многопоточной модели выполнения. Язык Java освобождает программиста от бремени управления памятью с помощью механизма сборки мусора. Поддержка многопоточной модели выполнения в языке C++ в настоящее время включена в состав стандартной библиотеки, однако в языке C она реализована только в виде сторонних библиотек. Но, несмотря на все это, создание многопоточных

программ на языке C, C++ или Java требует от программиста немалых усилий, чтобы обеспечить своевременное приобретение и освобождение ресурсов.

Все сложности, связанные с учетом ресурсов в языке Go, берут на себя компилятор и среда выполнения. Для управления памятью в Go имеется механизм сборки мусора, что избавляет от необходимости использовать «интеллектуальные» указатели или освобождать память вручную. А поддержка параллелизма в языке Go реализована в форме механизма взаимодействующих последовательных процессов (Communicating Sequential Processes, CSP), основанного на идеях специалиста в области теории вычислительных машин и систем Чарльза Энтони Ричарда Хоара (C. A. R. Hoare), благодаря которому во многих многопоточных программах на языке Go вообще отпадает необходимость блокировать доступ к ресурсам. Кроме того, в языке Go имеются так называемые *go-подпрограммы* (*goroutines*) – очень легковесные процессы, которых можно создать великое множество. Выполнение этих процессов автоматически будет распределяться по доступным процессорам и ядрам, что обеспечивает возможность более тонкого деления программ на параллельно выполняющиеся задачи, чем это позволяют другие языки программирования, основанные на потоках выполнения. Фактически поддержка параллелизма в языке Go реализована настолько просто и естественно, что при переносе однопоточных программ на язык Go часто обнаруживается возможность параллельного выполнения нескольких задач, ведущая к увеличению скорости выполнения и более оптимальному использованию машинных ресурсов.

Go – практичный язык, где во главу угла поставлены эффективность программ и удобство программиста. Например, встроенные и определяемые пользователем типы данных в языке Go существенно отличаются – операции с первыми из них могут быть значительно оптимизированы, что невозможно для последних. В Go имеются также два встроенных фундаментальных типа коллекций: *срезы* (*slices*) (фактически ссылки на массивы переменной длины) и *отображения* (*maps*) (словари, или хеши пар ключ/значение). Коллекции этих типов высокооптимизированы и с успехом могут использоваться для решения самых разных задач. В языке Go также поддерживаются указатели (это действительно компилирующий язык программирования – в нем отсутствует какая-либо виртуальная машина, снижающая производительность), что позволяет с непринужденностью

создавать собственные, весьма сложные типы данных, такие как сбалансированные двоичные деревья.

В то время как C поддерживает только процедурное программирование, а Java вынуждает программистов писать все программы в объектно-ориентированном стиле, Go позволяет использовать парадигму, наиболее подходящую для конкретной задачи. Go можно использовать как исключительно процедурный язык программирования, но он также обладает превосходной поддержкой объектно-ориентированного стиля программирования. Однако, как будет показано далее в книге, реализация объектно-ориентированной парадигмы в Go радикально отличается от реализации этой же парадигмы в таких языках, как C++, Java или Python. Она намного проще в использовании и значительно гибче.

Как и в языке C, в Go отсутствуют генерики (generics) (шаблоны, в терминологии C++), однако в Go имеется масса других возможностей, которые во многих случаях устраняют потребность в генериках. В языке Go отсутствует препроцессор и не используются подключаемые заголовочные файлы (что является еще одной причиной, объясняющей высокую скорость компиляции), поэтому в нем нет необходимости дублировать сигнатуры функций, как в C и C++. А благодаря отсутствию препроцессора семантика программы не может измениться незаметно для программиста, как это может произойти при небрежном обращении с директивами `#define` в языках C и C++.

Возможно, языки C++, Objective-C и Java создавались как улучшенные версии языка C (а последний – как улучшенная версия языка C++). В языке Go тоже можно усмотреть попытку создать улучшенную версию языка C, даже при том, что простой и ясный синтаксис Go больше напоминает язык Python – срезы и отображения в Go сильно напоминают списки и словари в Python. Однако по духу Go все-таки ближе к языку C, чем к любым другим языкам, и его можно считать попыткой устранить недостатки языка C, взять из него все самое лучшее и добавить множество новых возможностей, уникальных для Go.

Первоначально Go задумывался как язык системного программирования с высокой скоростью компиляции для разработки высокомасштабируемых программ, которые могли бы использовать преимущества распределенных систем и многоядерных компьютеров. В настоящее время область применения языка Go стала значительно шире первоначальной концепции, и сейчас он используется как высокопроизводительный язык программирования общего назначения, использовать который – одно удовольствие.

Структура книги

Глава 1 начинается с описания, как компилировать и запускать программы на языке Go. Затем в этой главе дается краткий обзор синтаксиса и возможностей языка Go, а также вводятся некоторые пакеты из стандартной библиотеки. В ней будут представлены и описаны пять коротких примеров, иллюстрирующих различные возможности языка Go. Эта глава написана так, чтобы сформировать представление о языке и вселить в читателя уверенность в необходимости освоения языка Go. (В этой главе также описывается, как получить и установить Go.)

Главы со 2 по 7 детально рассматривают язык Go. Три главы посвящены встроенным типам данных: в главе 2 рассказывается об идентификаторах, логических и числовых типах. В главе 3 рассматриваются строки. И в главе 4 – коллекции.

Глава 5 описывает и демонстрирует инструкции и управляющие конструкции языка Go. Она также рассказывает, как создавать и использовать собственные функции, и заканчивает главы, демонстрирующие создание на языке Go однопоточных программ в процедурном стиле.

Глава 6 показывает особенности объектно-ориентированного программирования на языке Go. Данная глава включает описание структур языка Go, используемых для объединения и встраивания (делегирования) значений, и интерфейсов, применяемых для определения абстрактных типов, а также демонстрирует, как в некоторых ситуациях добиться эффекта наследования. В главе будут представлены несколько законченных примеров с подробным описанием, чтобы помочь читателю разобраться в объектно-ориентированном стиле программирования на языке Go, который может существенно отличаться от привычного стиля.

Глава 7 охватывает механизмы параллельного выполнения задач в языке Go и приводит еще больше примеров, чем глава об объектно-ориентированном программировании, опять же чтобы помочь читателю лучше понять эти новые аспекты.

Глава 8 демонстрирует, как читать из файлов и записывать в них собственные и стандартные двоичные данные, текст, а также данные в формате JSON и XML. (Работа с текстовыми файлами коротко рассматривается в главе 1 и в нескольких последующих главах, потому что это позволяет приводить более практичные примеры и упражнения.)

Глава 9 завершает книгу. Она начинается с демонстрации импортирования и использования пакетов из стандартной библиотеки, а затем собственных и сторонних пакетов. В ней также рассказывается,

как документировать и тестировать собственные пакеты, измерять их производительность. Последний раздел главы является кратким обзором инструментов, предоставляемых компилятором `gc` и стандартной библиотекой Go.

Go – небольшой, но очень богатый и выразительный язык программирования (если измерять количеством синтаксических конструкций, концепций и идиом), поэтому книга получилась такой удивительно объемной. В ней с самого начала демонстрируются примеры, написанные в хорошем стиле, характерном для языка Go[1]. Разумеется, что при таком подходе некоторые приемы сначала демонстрируются и лишь потом разъясняются подробно. Читатель может быть уверен, что все необходимое обязательно будет разъясняться в книге (и, разумеется, будут даваться ссылки на пояснения, находящиеся в другом месте).

Go – очаровательный язык, и пользоваться им доставляет одно удовольствие. Синтаксис и идиомы языка просты в изучении, но в нем имеются несколько совершенно новых концепций, которые могут оказаться незнакомыми многим читателям. Данная книга пытается помочь читателю совершить концептуальный прорыв, особенно в области объектно-ориентированного и параллельного программирования на языке Go, что может занять недели, а то и месяцы у тех, в чьем распоряжении имеется только сухая документация.

Благодарности

Ни одна техническая книга, написанная мной, не обошлась без помощи других людей, и эта книга также не является исключением.

Я хотел бы выразить особую благодарность двум моим друзьям-программистам, не имевшим прежде опыта работы с языком Go: Жасмин Бланшетт (Jasmin Blanchette) и Трентону Шульцу (Trenton Schulz). Оба они на протяжении многих лет помогали мне в создании книг, и их отзывы помогли мне написать книгу, которая отвечала бы потребностям других программистов, начинающих изучение языка Go.

Также книга существенно выиграла благодаря отзывам одного из основных разработчиков Go – Найджела Тао (Nigel Tao). Я не всегда следовал его советам, но его мнение всегда имело большое значение и

[1] Единственное исключение составляют примеры использования каналов, которые в первых главах всегда объявляются как двунаправленные, хотя используются для передачи данных только в одном направлении. Корректное объявление направления обмена данными в каналах производится с главы 7.

помогло значительно улучшить не только примеры программного кода, но и текст.

Кроме того, я получал помощь и от других, включая Дэвида Бодди (David Boddie), начинающего программиста на языке Go, давшего ряд ценных советов. А также от разработчиков Go: Яна Ланса Тейлора (Ian Lance Taylor) и особенно от Расса Кокса (Russ Cox), решившего немало проблем с программным кодом и пониманием концепций, и давшего простые и ясные пояснения, способствовавшие повышению точности изложения технических деталей.

В процессе работы над книгой я неоднократно задавал вопросы в списке рассылки `golang-nuts` и всегда получал вдумчивые и полезные ответы от многих собеседников. Я также получал отзывы от читателей предварительного, «чернового» издания, опубликованного на сайте Safari Book Online, что повлекло добавление важных пояснений.

Итальянская компания (www.develer.com), занимающаяся разработкой программного обеспечения, в лице Джованни Баджо (Giovanni Bajo) любезно предоставила бесплатный хостинг для хранения репозитория Mercurial, обеспечив мне душевное спокойствие в течение длинного периода работы над этой книгой. Спасибо Лоренцо Манцини (Lorenzo Mancini), настроившему и сопровождавшему этот репозиторий для меня. Я также очень благодарен Антону Бауэрсу (Anton Bowers) и Бену Томпсону (Ben Thompson), предоставившим мне место для моего веб-сайта www.qtrac.eu на их веб-сервере в начале 2011 года.

Спасибо Расселу Уиндеру (Russel Winder) за статью о лицензиях на программное обеспечение, размещенную в его блоге www.russel.org.uk. Многие его идеи были заимствованы в приложении B.

И, как обычно, спасибо Джеффу Кингстону (Jeff Kingston), создателю неуклюжей типографской системы, которую я на протяжении многих лет использовал для набора всех своих книг и многих других трудов.

Особое спасибо моему выпускающему редактору Дебре Уильямс Коли (Debra Williams Cauley), успешно подготовившей книгу совместно с издателем и обеспечившей техническую поддержку и практическую помощь.

Спасибо также заведующему производственным отделом Анне Попик (Anna Popick), которая снова с успехом обеспечила руководство технологическим процессом, а также корректору Одри Дойл (Audrey Doyle), великолепно справившейся со своей работой.

Как всегда, я хочу поблагодарить мою супругу Андреа (Andrea) за ее любовь и поддержку.

1. Обзор в пяти примерах

В этой главе приводится серия из пяти примеров с подробными их описаниями. Несмотря на небольшой размер примеров, каждый из них (кроме «Hello Who?») имеет некоторую практическую ценность, а все вместе они представляют краткий обзор ключевых возможностей языка Go и некоторых его основных пакетов. (То, что в других языках называется модулями или библиотеками, в языке Go называется *пакетами*, а все пакеты, распространяемые вместе с Go, образуют *стандартную библиотеку* языка Go.) Цель этой главы – сформировать представление о языке и вселить в читателя уверенность в необходимости освоения языка Go. Не стоит волноваться, если какие-то синтаксические конструкции или идиомы останутся за рамками понимания, – все, что будет показано в этой главе, обязательно будет рассматриваться в последующих главах.

Обучение программированию на языке Go с применением специфических приемов требует определенного времени и усилий. Желающие заняться переносом программ с языков C, C++, Java, Python и др. на язык Go должны найти время на изучение Go, особенно его объектно-ориентированных возможностей и средств организации параллельных вычислений, в долгосрочной перспективе это позволит сэкономить время и силы. А желающие писать на языке Go собственные приложения добьются большего успеха, если максимально будут использовать все его возможности, поэтому они также должны потратить силы и время на изучение – позднее все затраты окупятся с лихвой.

1.1. Начало

Go – это компилирующий язык, а не интерпретирующий, поэтому программы, написанные на этом языке, имеют максимальную производительность. Компиляция выполняется очень быстро – намного быстрее, чем в некоторых других языках, особенно в сравнении с языками C и C++.

Документация по языку Go

По адресу golang.org находится официальный веб-сайт языка Go, где можно найти массу свежей документации по языку Go. По ссылке **Packages** (Пакеты) можно перейти к разделу с документацией ко всем пакетам из стандартной библиотеки Go и их исходными текстами, которые могут очень пригодиться, когда в документации обнаруживаются пробелы. Ссылка **References** => **Command Documentation** (Справочники => Документация к командам) ведет в раздел с документацией к программам, распространяемым вместе с Go (компиляторам, инструментам сборки и др.). По ссылке **References** => **Language Specification** (Справочники => Спецификация языка) можно перейти к разделу с неофициальной и весьма полной спецификацией языка Go. А по ссылке **Documents => Effective Go** (Документы => Эффективный Go) находится документ, описывающий многие приемы программирования на Go.

На веб-сайте также имеется «песочница» (с ограниченным набором возможностей), где можно вводить, компилировать и опробовать небольшие программы на языке Go. Данная возможность будет полезна начинающим для проверки непонятных синтаксических конструкций и изучения сложного пакета `fmt`, содержащего инструменты для работы с форматированным текстом, или пакета `regexp` – механизма регулярных выражений. Строка поиска на веб-сайте языка Go может использоваться только для поиска документации – если потребуется отыскать другие ресурсы, посвященные языку Go, посетите страницу go-lang.cat-v.org/go-search.

Документацию по языку Go можно также просматривать локально, например в веб-браузере. Для этого выполните команду `godoc`, передав ей аргумент, сообщающий, что она должна действовать как веб-сервер. Ниже показано, как выполнить эту команду в консоли Unix (`xterm`, `gnome-terminal`, `konsole`, `Terminal.app` и др.):

```
$ godoc -http=:8000
```

Или в консоли Windows (например, в окне **Command Prompt** (Командная строка) или **MS-DOS Prompt** (Сеанс MS-DOS)):

```
C:\>godoc -http=:8000
```

Номер порта здесь выбран произвольно. Если он уже занят – просто выберите другой. Здесь предполагается, что выполняемый файл `godoc` находится в каталоге, указанном в переменной `PATH`.

Для просмотра локальной документации откройте веб-браузер и введите адрес http://localhost:8000. На экране появится страница, внешним видом напоминающая главную страницу веб-сайта golang.org. Ссылка **Packages** (Пакеты) ведет в раздел документации к стандартной библиотеке Go, где также имеется документация к сторонним

пакетам, установленным в каталог GOROOT. Если в системе определена переменная окружения GOPATH (например, для локальных программ и пакетов), рядом со ссылкой **Packages** (Пакеты) появится ссылка к разделу с соответствующей документацией. (Переменные GOROOT и GOPATH обсуждаются ниже в этой главе, а также в главе 9.) С помощью команды godoc можно еще просматривать документацию для всего пакета в целом или для отдельного его элемента непосредственно в консоли. Например, команда godoc image NewRGBA выведет описание функции image.NewRGBA(), а команда godoc image/png – описание пакета image/png в целом.

Стандартный компилятор языка Go называется gc, а в состав его инструментов входят программы: 5g, 6g и 8g – для компиляции, 5l, 6l и 8l – для компоновки и godoc – для просмотра документации. (В Windows эти программы называются 5g.exe, 6l.exe и т. д.) Такие странные имена были даны в соответствии с соглашениями об именовании компиляторов, принятыми в операционной системе Plan 9, где цифра определяет аппаратную архитектуру (например, «5» – ARM, «6» – AMD-64, включая 64-битные процессоры Intel, и «8» – Intel 386.) К счастью, нет необходимости напрямую использовать эти инструменты благодаря наличию высокоуровневого инструмента сборки программ на языке Go – go, который автоматически выбирает нужный компилятор и компоновщик.

Все примеры из этой книги, доступные для загрузки на странице www.qtrac.eu/gobook.html, были проверены в Linux и Mac OS X с помощью gc, и в Windows с помощью компилятора версии Go 1. Разработчики Go предполагают обеспечить обратную совместимость с версией Go 1 во всех последующих версиях Go 1.x, поэтому описание в книге и примеры должны быть верными для всей серии 1.x. (Со временем, при обнаружении каких-либо несовместимостей, загружаемые примеры для книги будут обновляться в соответствии с последней версией Go, поэтому они могут отличаться от программного кода в книге.)

Чтобы загрузить и установить Go, откройте страницу golang.org/doc/install.html, где приводятся ссылки для загрузки и инструкции по установке. На момент написания этих строк версия Go 1 была доступна в виде двоичных и исходных файлов для FreeBSD 7+, Linux 2.6+, Mac OS X (Snow Leopard и Lion) и Windows 2000+ и во всех случаях для аппаратных архитектур Intel 386 and AMD-64. Для Linux имеется также поддержка архитектуры ARM. Предварительно собранные пакеты Go имеются для дистрибутива Ubuntu Linux, и

к моменту, когда вы будете читать эти строки, они могут появиться для других дистрибутивов Linux. Для начинающих изучать программирование на языке Go проще установить двоичную версию, чем собирать инструменты Go из исходных текстов.

Для программ, собираемых компилятором gc, действуют определенные соглашения об именовании. То есть программы, скомпилированные с помощью gc, могут быть скомпонованы только с внешними библиотеками, следующими тем же соглашениям, в противном случае необходимо использовать подходящий инструмент, устраняющий разногласия. В комплект Go входит инструмент cgo (golang.org/cmd/cgo), обеспечивающий возможность использования внешнего программного кода на языке C в программах на языке Go, кроме того, в Linux и BSD-системах имеется возможность использовать код на C и C++ с помощью инструмента SWIG (www.swig.org).

Помимо gc, имеется также компилятор gccgo. Это интерфейс к компилятору gcc (GNU Compiler Collection) для языка Go, который может быть задействован с компиляторами gcc, начиная с версии 4.6. Подобно gc, компилятор gccgo может быть доступен в некоторых дистрибутивах Linux в виде готовых пакетов. Инструкции по сборке и установке компилятора gccgo можно найти на странице golang.org/doc/gccgo_install.html.

1.2. Правка, компиляция и запуск

Программы на языке Go записываются в виде простого текста Юникода с использованием кодировки UTF-8[1]. Большинство современных текстовых редакторов обеспечивают эту поддержку автоматически, а некоторые наиболее популярные из них поддерживают даже подсветку синтаксиса для языка Go и автоматическое оформление отступов. Если ваш текстовый редактор не поддерживает Go, попробуйте ввести имя редактора в строке поиска на сайте Go, чтобы узнать, имеются ли для него расширения, обеспечивающие требуемую поддержку. Для удобства правки все ключевые слова и операторы языка Go записываются символами ASCII, однако идентификаторы в языке Go могут начинаться с любых алфавитных символов

[1] Некоторые текстовые редакторы для Windows (такие как Notepad (Блокнот)) не следуют рекомендациям стандарта Юникода и вставляют байты 0xEF, 0xBB, 0xBF в начало файлов с текстом в кодировке UTF-8. В этой книге предполагается, что файлы в кодировке UTF-8 не содержат этих байтов.

Юникода и содержать любые алфавитные символы Юникода или цифры. Благодаря этому программисты на Go свободно могут определять идентификаторы на своем родном языке.

Сценарии на языке Go

Одним из побочных эффектов высокой скорости компиляции программ на языке Go является возможность создания сценариев в Unix-подобных системах, начинающихся со строки `#!`. Для этого достаточно лишь установить подходящий инструмент, выполняющий компиляцию и запуск программы. На момент написания этих строк имелись два таких инструмента: `gonow` (github.com/kless/gonow) и `gorun` (wiki.ubuntu.com/gorun).

После установки `gonow` или `gorun` любую программу на языке Go можно оформить в виде сценария. Достигается это выполнением двух простых действий. Первое – добавить строку `#!/usr/bin/env gonow` или `#!/usr/bin/env gorun` в самое начало файла с расширением `.go`, содержащим функцию `main()` (в пакете `main`). Второе – дать файлу права на выполнение (например, командой `chmod +x`). Такие файлы могут компилироваться только инструментами `gonow` и `gorun`, потому что строка `#!` не является синтаксически допустимой строкой на языке Go.

При первом запуске команда `gonow` или `gorun` скомпилирует файл с расширением `.go` (очень быстро, разумеется) и запустит его. При последующих попытках перекомпиляция будет выполняться, только если исходный файл `.go` изменился с момента предыдущей компиляции. Это делает возможным написание на языке Go различных небольших вспомогательных программ, например для решения задач системного администрирования.

Чтобы получить представление, как писать, компилировать и выполнять программы на языке Go, начнем с классического примера «Hello World». Несмотря на небольшой размер, программа будет выглядеть чуть сложнее, чем обычно. Но для начала обсудим компиляцию и запуск программы, а затем, в следующем разделе, перейдем к детальному изучению исходного программного кода в файле `hello/hello.go`, использующего некоторые базовые идеи и особенности языка Go.

Все примеры для этой книги доступны на странице www.qtrac.eu/gobook.html в виде архива каталога `goeg`. Поэтому полный путь к файлу `hello.go` (предполагается, что архив с примерами распакован непосредственно в домашний каталог, хотя его можно распаковать в любой другой каталог) будет иметь вид `$HOME/goeg/src/hello/hello.go`. При ссылке на имена файлов в этой книге всегда

будет предполагаться наличие первых трех компонентов пути, то есть в данном случае путь к файлу выглядит как `hello/hello.go`. (Разумеется, пользователи Windows должны читать символ «/» как «\» и использовать имя каталога, куда были распакованы примеры, например: `C:\goeg` или `%HOMEPATH%\goeg`.)

Если Go был установлен из двоичного дистрибутива или собран из исходных текстов и установлен с привилегиями пользователя root или Administrator, необходимо создать хотя бы одну переменную окружения, `GOROOT`, содержащую путь к каталогу установки Go, а в переменную `PATH` включить путь `$GOROOT/bin` или `%GOROOT%\bin`. Чтобы убедиться, что установка Go была выполнена правильно, можно выполнить следующую команду консоли Unix (`xterm`, `gnome-terminal`, `konsole`, `Terminal.app` и др.):

```
$ go version
```

Или в консоли Windows (например, в окне **Command Prompt** (Командная строка) или **MS-DOS Prompt** (Сеанс MS-DOS)):

```
C:\>go version
```

Если в консоли появится сообщение «command not found» (команда не найдена) или «'go' is not recognized...» (команда go не опознана), это означает, что путь к каталогу установки Go не был включен в переменную `PATH`. Простейший способ решить эту проблему в Unix-подобных системах (включая Mac OS X) – установить значения переменных окружения в файле `.bashrc` (или в эквивалентном ему, если используется другая командная оболочка). Например, файл `.bashrc` у автора содержит следующие строки:

```
export GOROOT=$HOME/opt/go
export PATH=$PATH:$GOROOT/bin
```

Естественно, конкретные значения следует установить в соответствии со своей системой. (И, разумеется, делать это необходимо только в случае неудачной попытки выполнить команду `go version`.)

Для Windows одно из решений заключается в том, чтобы создать пакетный файл, настраивающий окружение Go, и выполнять его при каждом запуске консоли для программирования на языке Go. Однако намного удобнее один раз настроить переменные окружения в панели управления. Для этого щелкните на кнопке **Start**

(Пуск) (с логотипом Windows), выберите пункт меню **Control Panel** (Панель управления), затем пункт **System and Security** (Система и безопасность), потом **System** (Система), далее **Advanced system settings** (Дополнительные параметры системы) и в диалоге **System Properties** (Свойства системы) щелкните на кнопке **Environment Variables** (Переменные окружения), затем на кнопке **New...** (Создать) и добавьте переменную с именем `GOROOT` и соответствующим значением, таким как `C:\Go`. В том же диалоге отредактируйте значение переменной окружения `PATH`, добавив в конец текст `;C:\Go\bin` – начальная точка с запятой имеют важное значение! В обоих случаях замените компонент пути `C:\Go` на фактический путь к каталогу установки Go, если он отличается от `C:\Go`. (Опять же, делать это необходимо только в случае неудачной попытки выполнить команду `go version`.)

С этого момента будет предполагаться, что язык Go установлен и путь к его каталогу `bin`, содержащему все инструменты Go, включен в переменную окружения `PATH`. (Чтобы новые настройки вступили в силу, может потребоваться открыть новое окно консоли.)

Сборка программ на языке Go выполняется в два этапа: компиляция и компоновка[1]. Оба этапа выполняются инструментом `go`, который не только собирает локальные программы и пакеты, но также способен загружать, собирать и устанавливать сторонние программы и пакеты.

Чтобы обеспечить сборку локальных программ и пакетов с помощью инструмента `go`, необходимо выполнить три обязательных условия. Первое: каталог `bin` с инструментами языка Go (`$GOROOT/bin` или `%GOROOT%\bin`) должен находиться в пути поиска `PATH`. Второе: в дереве каталогов должен существовать каталог `src` для хранения исходных текстов локальных программ и пакетов. Например, примеры для книги распаковываются в каталоги `goeg/src/hello`, `goeg/src/bigdigits` и т. д. Третье: путь к каталогу, *вмещающему* каталог `src`, должен быть включен в переменную окружения `GOPATH`. Так, чтобы собрать пример `hello` с помощью инструмента `go`, необходимо выполнить следующие операции:

[1] Поскольку эта книга предполагает, что компиляция выполняется с помощью компилятора `gc`, читатели, использующие `gccgo`, должны выполнять компиляцию и компоновку в соответствии с инструкциями на странице golang.org/doc/gccgo_install.html. Аналогично читатели, использующие другие компиляторы, должны выполнять компиляцию и компоновку в соответствии с инструкциями для их компилятора.

```
$ export GOPATH=$HOME/goeg
$ cd $GOPATH/src/hello
$ go build
```

В Windows эти операции выполняются практически так же:

```
C:\>set GOPATH=C:\goeg
C:\>cd %gopath%\src\hello
C:\goeg\src\hello>go build
```

В обоих случаях предполагается, что переменная PATH включает путь $GOROOT/bin или %GOROOT%\bin. После сборки программы инструментом go ее можно запустить. По умолчанию выполняемый файл получает имя каталога, в который он помещается (например, hello – в Unix-подобных системах и hello.exe – в Windows). Запуск программы выполняется как обычно.

```
$ ./hello
Hello World!
```

Или:

```
$ ./hello Go Programmers!
Hello Go Programmers!
```

В Windows запуск выполняется похожим способом:

```
C:\goeg\src\hello>hello Windows Go Programmers!
Hello Windows Go Programmers!
```

В примерах выше жирным шрифтом выделен текст, который должен вводиться вручную. Здесь также предполагается, что строка приглашения к вводу имеет вид $, но на самом деле это не имеет никакого значения (она может иметь вид, например, C:\>).

Обратите внимание на *отсутствие* необходимости компилировать или явно компоновать какие-либо другие пакеты (хотя в исходном файле hello.go, как будет показано ниже, используются три пакета из стандартной библиотеки). Это еще одна причина, объясняющая высокую скорость компиляции программ на языке Go.

Если бы потребовалось скомпилировать несколько программ, было бы удобнее, если бы все выполняемые файлы помещались в один

каталог, путь к которому включен в переменную PATH. К счастью, инструмент go поддерживает такую возможность:

```
$ export GOPATH=$HOME/goeg
$ cd $GOPATH/src/hello
$ go install
```

Или то же самое в Windows:

```
C:\>set GOPATH=C:\goeg
C:\>cd %gopath%\src\hello
C:\goeg\src\hello>go install
```

Команда go install делает то же самое, что и команда go build, но помещает выполняемые файлы в стандартный каталог ($GOPATH/bin или %GOPATH%\bin). То есть, если добавить путь ($GOPATH/bin или %GOPATH%\bin) в переменную PATH, *все* программы на языке Go будут устанавливаться в каталог, путь к которому включен в переменную PATH.

Помимо примеров из книги, многие пожелают писать на языке Go свои программы и пакеты и хранить их в отдельных каталогах. Обеспечить такую возможность можно простым включением в переменную окружения GOPATH двух (или более) путей к каталогам, разделенных двоеточием (точкой с запятой, в Windows). Например: export GOPATH=$HOME/app/go:$HOME/goeg или SET GOPATH=C:\app\go;C:\goeg[1]. В данном случае необходимо будет помещать все исходные тексты программ и пакетов в каталог $HOME/app/go/src или C:\app\go\src. То есть при создании программы с именем myapp ее исходный файл с расширением .go должен находиться в каталоге $HOME/app/go/src/myapp или C:\app\go\src\myapp. И если для сборки программы будет использоваться команда go install и при этом программа будет находиться в каталоге, путь к которому включен в переменную GOPATH, содержащую два или более каталога, выполняемый файл будет сохранен в соответствующем каталоге bin.

Естественно, слишком утомительно настраивать или экспортировать переменную GOPATH каждый раз, когда потребуется собрать программу, поэтому лучше определить эту переменную окружения на

[1] С этого момента практически всегда будут демонстрироваться команды только в стиле ОС Unix и предполагаться, что программисты, использующие ОС Windows, смогут мысленно преобразовать их в команды Windows.

постоянной основе. Сделать это можно, включив определение переменной GOPATH в файл .bashrc (или подобный ему) в Unix-подобных системах (см. файл gopath.sh в примерах к книге). В Windows то же самое можно сделать, либо написав пакетный файл (см. файл gopath.bat в примерах к книге), либо добавив определение переменной в системные переменные окружения: щелкните на кнопке **Start** (Пуск) (с логотипом Windows), выберите пункт меню **Control Panel** (Панель управления), затем выберите пункт **System and Security** (Система и безопасность), далее **System** (Система), потом **Advanced system settings** (Дополнительные параметры системы) и в диалоге **System Properties** (Свойства системы) щелкните на кнопке **Environment Variables** (Переменные окружения), затем на кнопке **New...** (Создать) и добавьте переменную с именем GOROOT и соответствующим значением, таким как C:\goeg или C:\app\go;C:\goeg.

Утилита go является стандартным инструментом сборки программ на языке Go. Тем не менее для тех же целей с успехом можно использовать утилиту make, другие подобные средства или альтернативные инструменты, предназначенные для сборки программ на языке Go, а также расширения для популярных интегрированных сред разработки (Integrated Development Environments, IDE), таких как Eclipse и Visual Studio.

1.3. Hello кто?

Теперь, когда стало понятно, как собрать программу hello, обратимся к исходным текстам. Не волнуйтесь, если что-то останется за рамками понимания, – все, что будет показано в этой главе (и многое другое!), будет подробно описываться в последующих главах. Ниже приводится полный листинг программы hello (в файле hello/hello.go):

```
// hello.go
package main
import ( ❶
    "fmt"
    "os"
    "strings"
)
func main() {
    who := "World!" ❷
    if len(os.Args) > 1 { /* os.Args[0] - имя команды «hello» или «hello.exe» */ ❸
```

```
        who = strings.Join(os.Args[1:], " ") ❹
    }
    fmt.Println("Hello", who) ❺
}
```

Комментарии в языке Go оформляются в стиле языка C++: однострочные комментарии, заканчивающиеся в конце строки, начинаются с символов //, а блочные комментарии, занимающие несколько строк, заключаются в символы /* ... */. Обычно в программах на языке Go используются однострочные комментарии, включая комментарии, используемые для исключения из программы фрагментов программного кода во время отладки[1].

Любой фрагмент программного кода на языке Go должен быть включен в пакет, а каждая программа должна иметь пакет `main` с функцией `main()`, которая является точкой входа в программу, то есть с функцией, выполняющейся в первую очередь. В действительности пакеты на языке Go могут также иметь функцию `init()`, которая выполняется перед функцией `main()`, как будет показано ниже (§1.7). Подробнее об этом будет рассказываться далее (§5.6.2). Обратите внимание, что здесь между именем пакета и именем функции нет никакого конфликта.

Язык Go оперирует в терминах *пакетов*, а не файлов. То есть пакет можно разбить на любое количество файлов, и если все они будут иметь одинаковое объявление пакета, с точки зрения языка Go все они будут являться частями одного и того же пакета, как если бы все их содержимое находилось в единственном файле. Естественно, точно так же всю функциональность приложения можно распределить по нескольким пакетам, чтобы обеспечить модульный принцип построения, как будет показано в главе 9.

Инструкция `import` (❶ в листинге выше) импортирует три пакета из стандартной библиотеки. Пакет `fmt` содержит функции форматирования текста и чтения форматированного текста (§3.5), пакет `os` содержит платформонезависимые системные переменные и функции, а пакет `strings` – функции для работы со строками (§3.6.1).

Фундаментальные типы данных в языке Go поддерживают привычные операторы (например, + – для сложения чисел и

[1] В листингах примеров будет использоваться прием подсветки синтаксиса и к отдельным строкам будут добавляться числа (❶, ❷, ...), чтобы проще было ссылаться на них в тексте. Ни один из этих элементов не является частью языка Go.

конкатенации строк), а стандартная библиотека Go добавляет дополнительные пакеты функций для работы с фундаментальными типами, такие как импортированный здесь пакет `strings`. Кроме того, имеется возможность определять пользовательские типы данных, опираясь на фундаментальные типы, и предусматривать собственные методы, то есть функции, для работы с ними. (Коротко об этом будет рассказываться в §1.5 ниже, а подробно эта тема будет обсуждаться в главе 6.)

Внимательный читатель, возможно, заметил, что в программе отсутствуют точки с запятой, импортируемые пакеты не отделяются друг от друга запятыми и условное выражение в инструкции `if` не требуется заключать в круглые скобки. В языке Go блоки программного кода, включая тела функций и управляющих конструкций (например, инструкций `if` и циклов `for`), заключаются в фигурные скобки. Отступы используются исключительно для удобства человека. Технически инструкции в языке Go должны отделяться друг от друга точками с запятой, но они автоматически добавляются компилятором, поэтому в них нет необходимости, если в одной строке не располагаются несколько инструкций. Отсутствие необходимости вставлять точки с запятой, небольшое количество ситуаций, когда требуется вставлять запятые и фигурные скобки, делают программы на языке Go более удобочитаемыми и уменьшают объем ввода с клавиатуры.

Функции и методы в языке Go определяются с помощью ключевого слова `func`. Функция `main()` в пакете `main` всегда имеет одну и ту же сигнатуру – она не имеет аргументов и ничего не возвращает. Когда функция `main.main()` завершается, одновременно с ней завершается выполнение программы, и она возвращает операционной системе значение `0`. Естественно, имеется возможность в любой момент завершить работу программы и вернуть любое значение, как будет показано далее (§1.4).

Первая инструкция в функции `main()` (❷ в листинге выше, где используется оператор :=) в терминологии языка Go называется *сокращенным объявлением переменной*. Такие инструкции одновременно объявляют и инициализируют переменные. Кроме того, в подобных инструкциях нет необходимости объявлять тип переменной, потому что компилятор Go автоматически определит его по присваиваевому значению. Таким образом, в данном случае объявляется переменная с именем `who` типа `string`, и, как следствие строгой типизации в языке Go, далее переменной `who` могут присваиваться только строковые значения.

Как и во многих других языках программирования, инструкция `if` проверяет условие, в данном случае – количество аргументов командной строки, и если оно удовлетворяется, выполняет соответствующий блок программного кода, заключенный в фигурные скобки. В этой главе ниже будет показан более сложный синтаксис инструкции `if` (§1.6), а подробнее о нем будет рассказываться в главе 5 (§5.2.1).

Переменная `os.Args` – это срез со строками строк (❸ в листинге выше). Массивы, срезы и другие типы коллекций рассматриваются в главе 4 (§4.2). Пока достаточно знать, что длину среза можно определить с помощью встроенной функции `len()`, а обращаться к его элементам можно с помощью оператора индексирования `[]`, используя подмножество синтаксиса языка Python. В частности, выражение `slice[n]` вернет *n*-й элемент среза (отсчет элементов начинается с нуля), а выражение `slice[n:]` – другой срез, содержащий с *n*-го по последний элемент из первого среза. Полный синтаксис языка Go в этой области будет представлен в главе, посвященной коллекциям. В случае с переменной `os.Args` срез всегда должен содержать хотя бы одну строку (имя программы) в элементе с индексом `0`. (В языке Go отсчет элементов всегда начинается с `0`.)

Если пользователь передаст программе один или более аргументов командной строки, условие в инструкции `if` выполнится, и в переменную `who` запишутся все аргументы, объединенные в одну строку (❹ в листинге выше). В данном случае используется оператор присваивания (`=`), поскольку при использовании оператора сокращенного объявления переменной (`:=`) будет объявлена и инициализирована *новая* переменная `who`, область видимости которой будет ограничена телом инструкции `if`. Функция `strings.Join()` принимает срез со строками и строку-разделитель (можно указать пустую строку, `""`) и возвращает единственную строку, содержащую все строки из среза, разделенные строкой-разделителем. В данном случае в качестве строки-разделителя используется единственный пробел.

Наконец, последняя инструкция (❺ в листинге выше) выводит слово «Hello», пробел, строку из переменной `who` и символ перевода строки. Пакет `fmt` имеет множество разновидностей функции вывода, некоторые, такие как `fmt.Println()`, просто выводят все, что им передается, другие, такие как `fmt.Printf()`, позволяют использовать спецификаторы формата, обеспечивающие тонкое управление форматированием. Подробно функции вывода рассматриваются в главе 3 (§3.5).

Представленная здесь программа `hello` демонстрирует намного больше особенностей языка, чем обычно делают подобные программы. Последующие примеры построены в том же духе – они демонстрируют более широкий круг возможностей, оставаясь предельно короткими. Основная идея этой главы состоит в том, чтобы дать начальные представления о языке, а также научить собирать, выполнять и проводить эксперименты с простыми программами на языке Go, одновременно знакомя с различными возможностями языка Go. И конечно же все, представленное в этой главе, будет подробно разъясняться в последующих главах.

1.4. Большие цифры – двумерные срезы

Программа `bigdigits` (в файле `bigdigits/bigdigits.go`) читает число, введенное в командной строке (в виде строки), и выводит то же число «большими» цифрами. В прошлом веке, в организациях, где множество людей совместно пользовалось одним высокоскоростным принтером, обычной практикой было для каждого задания печатать титульные листы, содержащие некоторую идентификационную информацию, такую как имя пользователя, имя печатаемого файла, с использованием подобного приема.

Изучение программного кода будет разделено на три этапа: сначала будет рассмотрена инструкция импортирования, затем объявление статических данных и потом – собственно обработка. Но прежде рассмотрим результаты запуска программы, чтобы иметь представление, как она действует:

```
$ ./bigdigits 290175493
 222     9999     000      1   77777  55555     4     9999    333
2   2  9   9   0   0   11       7  5         44    9   9  3   3
   2   9   9  0     0   1      7   5        4 4    9   9       3
  2     9999  0     0   1     7     555    4  4     9999     33
 2         9  0     0   1    7          5  444444      9       3
2          9   0   0    1   7       5   5     4        9  3   3
22222      9    000    111  7       555      4        9   333
```

Каждая цифра представлена срезом со строками, в котором все цифры представлены срезом срезов со строками. Прежде чем перейти к данным, взгляните, как можно объявить и инициализировать одномерные срезы со строками и числами:

```
longWeekend := []string{"Friday", "Saturday", "Sunday", "Monday"}
var lowPrimes = []int{2, 3, 5, 7, 11, 13, 17, 19}
```

Объявление среза имеет вид []Тип, а если необходимо сразу инициализировать его, вслед за объявлением типа можно в фигурных скобках указать список значений соответствующего типа, разделенных запятыми. В обоих примерах допустимо было бы использовать один и тот же синтаксис объявления, но для среза lowPrimes была выбрана более длинная форма, чтобы показать синтаксические различия, а также по причине, описываемой чуть ниже. Поскольку в качестве Типа можно указать срез, появляется простая возможность создания многомерных коллекций (срезы срезов и т. д.).

Программе bigdigits требуется импортировать всего четыре пакета.

```
import (
    "fmt"
    "log"
    "os"
    "path/filepath"
)
```

Пакет fmt содержит функции форматирования текста и чтения форматированного текста (§3.5). Пакет log предоставляет функции журналирования. Пакет os – платформонезависимые системные переменные и функции, включая переменную os.Args типа []string (срез со строками), хранящую аргументы командной строки. И пакет filepath из пакета path предоставляет функции для работы с именами файлов и путями в файловой системе платформонезависимым способом. Обратите внимание, что для пакетов, логически включенных в другие пакеты, при обращении в программном коде указывается только последний компонент их имени (в данном случае filepath).

В программе bigdigits необходимо объявить двумерную коллекцию данных (срез срезов со строками). Ниже показано объявление среза со строками для цифры 0, растянутое так, чтобы продемонстрировать, как строки в срезе соответствуют строкам в выводе программы, за которым следуют объявления срезов со строками для других цифр, при этом данные для цифр с 3 по 8 опущены.

```
var bigDigits = [][]string{
```

```
    {"  000 ",
     " 0   0 ",
     "0     0",
     "0     0",
     "0     0",
     " 0   0 ",
     "  000 "},
    {" 1 ", "11 ", " 1 ", " 1 ", " 1 ", " 1 ", "111"},
    {" 222 ", "2   2", "    2 ", "  2  ", " 2   ", "2    ", "22222"},
    // ... с 3 по 8 ...
    {" 9999", "9   9", "9   9", " 9999", "    9", "    9", "    9"},
}
```

Для объявления переменных за пределами функций или методов можно не применять оператор `:=` , но тот же эффект можно получить, используя длинную форму объявления (с ключевым словом `var`) и оператор присваивания (`=`), как было сделано при объявлении переменной `bigDigits` (и выше, в примере объявления переменной `lowPrimes`). При этом не обязательно объявлять тип переменной `bigDigits`, поскольку он автоматически определяется компилятором Go из выражения присваивания.

Кроме того, компилятор сам может подсчитать количество элементов, поэтому нет необходимости указывать размерности среза срезов. Одна из замечательных особенностей языка Go заключается в поддержке *составных литералов*, сконструированных с применением фигурных скобок, благодаря этому нет необходимости объявлять переменную в одном месте, а наполнять ее данными в другом, если, конечно, у вас не возникнет такого желания.

Функция `main()`, которая читает аргументы командной строки и использует их для формирования вывода, занимает всего 20 строк.

```
func main() {
    if len(os.Args) == 1 { ❶
        fmt.Printf("usage: %s <whole-number>\n", filepath.Base(os.Args[0]))
        os.Exit(1)
    }
    stringOfDigits := os.Args[1]
    for row := range bigDigits[0] { ❷
        line := ""
        for column := range stringOfDigits { ❸
            digit := stringOfDigits[column] - '0' ❹
            if 0 <= digit && digit <= 9 { ❺
```

```
                line += bigDigits[digit][row] + "  " ❻
            } else {
                log.Fatal("invalid whole number")
            }
        }
        fmt.Println(line)
    }
}
```

Программа начинается с проверки наличия аргументов командной строки. Если аргументы отсутствуют, вызов `len(os.Args)` вернет `1` (напомню, что элемент `os.Args[0]` хранит имя программы, поэтому длина среза не может быть меньше 1) и условие в первой инструкции `if` (❶ в листинге выше) будет удовлетворено. В этом случае будет выведено сообщение о порядке использования программы с помощью функции `fmt.Printf()`, принимающей спецификаторы формата `%`, напоминающие спецификаторы, поддерживаемые функцией *printf()* в языках C/C++ или оператор `%` в языке Python. (Полное описание приводится в §3.5.)

Пакет `path/filepath` предоставляет функции для выполнения операций со строками путей. Например, функция `filepath.Base()` возвращает базовое имя (то есть имя файла) в указанной строке пути. После вывода сообщения программа завершается вызовом функции `os.Exit()` и возвращает операционной системе значение `1`. В Unix-подобных системах возвращаемое значение 0 свидетельствует об успешном завершении, а ненулевое значение – об ошибке.

Вызов функции `filepath.Base()` иллюстрирует одну замечательную особенность языка Go: при ссылке на импортированный пакет, будь это пакет верхнего уровня или логически вложенный в другой пакет (например, `path/filepath`), всегда используется только последний компонент его полного имени (например, `filepath`). Кроме того, чтобы избежать конфликтов имен, пакетам можно присваивать локальные имена, о чем подробно рассказывается в главе 9.

Если пользователь передал программе хотя бы один аргумент, первый из них копируется в переменную `stringOfDigits` (типа `string`). Чтобы преобразовать число, переданное пользователем, в последовательность больших цифр, необходимо выполнить итерации по всем рядам в срезе `bigDigits` и вывести первые (верхние) строки образов каждой из цифр в числе, затем вторые и т. д. Предполагается, что срезы в `bigDigits` содержат одинаковое количество рядов, поэтому цикл строится с учетом количества рядов в образе

первой цифры. Цикл `for` в языке Go может иметь различные формы в зависимости от преследуемых целей. Здесь (❷ и ❸ в листинге выше) используются циклы `for ...range`, возвращающие индекс каждого элемента в указанном срезе.

Циклы обхода рядов и столбцов в образах цифр можно было бы записать так:

```
for row := 0; row < len(bigDigits[0]); row++ {
    line := ""
    for column := 0; column < len(stringOfDigits); column++ {
        ...
```

Эта форма знакома программистам на C, C++ и Java и вполне допустима в языке Go[1]. Однако форма записи `for ...range` короче и удобнее. (Подробнее о циклах в языке Go рассказывается в §5.3.)

В каждой итерации по рядам переменной `line` присваивается пустая строка. Затем начинаются итерации по столбцам (то есть по символам) в строке `stringOfDigits`, полученной от пользователя. Строки в языке Go хранят символы в кодировке UTF-8, поэтому теоретически символы могут быть представлены двумя и более байтами. В данном случае это не вызывает проблем, потому что программа обрабатывает только цифры `0`, `1`, ..., `9`, каждая из которых в кодировке UTF-8 представлена единственным байтом с тем же значением, что и в 7-битной кодировке ASCII. (Как выполнять итерации по символам в строках независимо от количества байт в них, будет показано в главе 3.)

При обращении к определенной позиции в строке по индексу возвращается *байт*, хранящийся в этой позиции. (В языке Go тип `byte` является синонимом типа `uint8`.) После извлечения байта из аргумента командной строки из него вычитается значение байта, соответствующего символу 0, чтобы получить его числовое представление (❹ в листинге выше). В кодировке UTF-8 (и 7-битной кодировке ASCII) символу `'0'` соответствует десятичное значение 48, символу `'1'` – значение 49 и т. д. Поэтому для символа `'3'` (значение

[1] В отличие от C, C++ и Java, в языке Go операторы `++` и `--` могут использоваться только как инструкции, но не как выражения. Кроме того, они могут применяться лишь в постфиксной форме. Это предотвращает появление проблем, связанных с неправильным порядком вычислений, то есть в Go нельзя записать выражения, такие как `f(i++)` и `a[i] = b[++i]`.

51), например, в результате вычитания '3' - '0' (то есть 51 – 48) будет получено целое число 3 (типа byte).

Для определения литералов символов в языке Go используются апострофы (одиночные кавычки), при этом литералы символов интерпретируются как целочисленные значения, совместимые со всеми целочисленными типами в языке Go. Строгое соблюдение типов в языке Go не позволяет, например, сложить значения типов int32 и int16 без явного их преобразования, но числовые константы и литералы обретают тип в зависимости от контекста их использования, поэтому в данном случае литерал '0' интерпретируется как значение типа byte.

Если значение переменной digit (типа byte) попадает в указанный диапазон (❺ в листинге выше), соответствующая ему строка добавляется в переменную line. (В инструкции if константы 0 и 9 интерпретируются как значения типа byte в соответствии с типом переменной digit, но если бы переменная digit имела другой тип, например int, константы интерпретировались бы как значения этого типа.) Строки в языке Go являются неизменяемыми значениями (то есть они не могут изменяться), тем не менее Go поддерживает простой и удобный оператор добавления +=. (При выполнении он замещает оригинальную строку.) Кроме того, поддерживается также оператор конкатенации +, возвращающий новую строку, являющуюся результатом объединения строковых операндов слева и справа. (Тип string подробно рассматривается в главе 3.)

Для извлечения строки (❻ в листинге выше) выполняется обращение к срезу внутри bigDigits, соответствующему цифре, а затем внутри него – к требуемому ряду (строке).

Если значение переменной digit оказывается вне указанного диапазона (например, когда stringOfDigits содержит нецифровые символы), вызывается функция log.Fatal() с текстом сообщения об ошибке. Эта функция выводит дату, время и сообщение в os.Stderr, если явно не было указано другое место для вывода, и вызывает os.Exit(1), чтобы завершить программу. Существует также функция log.Fatalf(), которая делает то же самое, но принимает спецификаторы формата %. Функция log.Fatal() не использовалась в первой инструкции if (❶ в листинге выше), потому что в сообщении о порядке использования программы нет необходимости указывать дату и время, которые log.Fatal() выводит автоматически.

Как только в переменную line будут добавлены все строки для заданного ряда образа числа, она выводится вызовом функции fmt.

`Println()`. В данном примере выводятся семь строк-рядов, потому что каждая цифра в срезе `bigDigits` представлена семью строками.

И наконец, следует отметить, что порядок объявлений и определений в общем случае не имеет большого значения. Поэтому в файле `bigdigits/bigdigits.go` можно было бы объявить переменную `bigDigits` как до, так и после функции `main()`. В данном случае первой следует функция `main()`, потому что в книжных примерах предпочтительнее, когда рассматриваемые элементы следуют в порядке сверху вниз.

Первые два примера охватывают массу основных понятий. Но оба они не демонстрируют ничего нового, что было бы незнакомо для имеющих опыт работы с другими языкам программирования, разве что немного отличающийся синтаксис. Следующие три примера выходят за рамки привычного комфорта и иллюстрируют особенности, характерные для языка Go, такие как определение пользовательских типов, обработка файлов (включая обработку ошибок) и функций как значений, и параллельное программирование с применением go-подпрограмм и каналов обмена данными.

1.5. Стек – пользовательские типы данных с методами

Хотя язык Go и поддерживает объектно-ориентированное программирование, в нем отсутствуют такие понятия, как классы и наследование (отношения типа «является»). В языке Go поддерживается возможность определения пользовательских типов и чрезвычайно простой способ агрегации типов (отношения типа «имеет»). Кроме того, в языке Go обеспечивается возможность полного отделения типов данных от их поведения и поддерживается *динамическая* (или так называемая утиная) *типизация*. Динамическая (утиная) типизация – мощный механизм абстракций, позволяющий обрабатывать значения (например, передаваемые функциям) с помощью предоставляемых ими методов, независимо от их фактических типов. Термин «утиная типизация» родился из фразы: «Если это ходит как утка и крякает как утка, значит, это утка». Все это дает более мощную и гибкую альтернативу классам и наследованию, но требует от привыкших к использованию традиционных подходов существенного пересмотра понятий, чтобы осознать все преимущества объектно-ориентированной модели в языке Go.

Данные в языке Go представляются с использованием встроенных фундаментальных типов, обозначаемых такими ключевыми словами, как `bool`, `int` и `string`, или составных типов, обозначаемых ключевым словом `struct`[1]. Пользовательские типы в языке Go опираются на фундаментальные типы, на структуры или другие пользовательские типы. (Простые примеры таких типов будут показаны далее в этой главе, в §1.7.)

Go поддерживает как именованные, так и неименованные пользовательские типы. Неименованные типы с одинаковой структурой можно использовать взаимозаменяемо, однако они не могут иметь методы. (Более подробно эта тема обсуждается в §6.4.) Любой именованный пользовательский тип может иметь методы, и эти методы составляют интерфейс типа. Именованные пользовательские типы, даже с одинаковой структурой, не могут использоваться взаимозаменяемо. (Далее в этой книге под термином «пользовательские типы» будут подразумеваться *именованные* пользовательские типы, если явно не будет указано иное.)

Интерфейс – это тип, имеющий формальное объявление и определяющий некоторый *набор методов*. Интерфейсы являются абстрактными типами и не позволяют создавать их экземпляры. Конкретный тип (то есть не интерфейс), имеющий методы, определяемые интерфейсом, реализует этот интерфейс. То есть значения такого конкретного типа могут использоваться как значения типа интерфейса, так и собственного, фактического типа. Тем не менее для реализации методов, определяемых интерфейсом, не требуется формально определять связь между интерфейсом и конкретным типом. Для пользовательского типа достаточно иметь реализацию методов, определяемых интерфейсом, чтобы удовлетворять этому интерфейсу. И конечно же тип может удовлетворять требованиям нескольких интерфейсов, реализуя методы всех этих интерфейсов.

Пустой интерфейс (то есть интерфейс без методов) объявляется как `interface{}`[2]. Так как пустой интерфейс вообще не предъявляет никаких требований (поскольку он не определяет ни одного метода), он может использоваться для ссылки на любое значение (подобно нетипизированному указателю), независимо от того, какой

[1] В отличие от C++, структуры в языке Go не являются замаскированными классами. Например, структуры в языке Go поддерживают агрегирование и делегирование, но не поддерживают наследование.

[2] Пустой интерфейс в языке Go может играть ту же роль, что и ссылка `Object` в Java или `void*` в C/C++.

тип имеет это значение. (Подробнее о ссылках и указателях в языке Go рассказывается в §4.1.) В данном случае мы говорим о типах и значениях, а не о классах и объектах или экземплярах (поскольку в языке Go отсутствует понятие классов).

Параметры функций и методов могут иметь любой тип, встроенный или пользовательский, или тип любого интерфейса. В последнем случае функция получит параметр, который может быть интерпретирован, например, так: «значение, которое позволяет читать данные», независимо от фактического типа этого значения. (Эта особенность будет показана на практике чуть ниже, в §1.6.)

В главе 6 все эти особенности будут рассматриваться более подробно, и там будет представлено достаточно большое количество примеров, чтобы можно было понять эту идею. А пока рассмотрим простой пользовательский тип – стек. Сначала посмотрим, как создавать и использовать значения этого типа, а затем перейдем к его реализации.

Начнем с изучения вывода простой тестовой программы `stacker`:

```
$ ./stacker
81.52
[pin clip needle]
-15
hay
```

Каждый элемент выталкивается из стека и выводится в отдельной строке.

Простая тестовая программа, которая вывела эти строки, находится в файле `stacker/stacker.go`. Ниже приводится список импортируемых ею пакетов:

```
import (
    "fmt"
    "stacker/stack"
)
```

Пакет `fmt` является частью стандартной библиотеки Go, а пакет `stack` – локальный, входящий в состав приложения `stacker`. Поиск программ на языке Go или импортируемых пакетов сначала выполняется в каталогах, включенных в переменную окружения `GOPATH`, а затем включенных в переменную `GOROOT`. В данном случае исходный программный код находится в файле `$HOME/goeg/src/stacker/`

stacker.go, а пакет stack – в файле $HOME/goeg/src/stacker/stack/stack.go. Утилита go соберет программу и пакет, при условии что переменная GOPATH содержит каталог $HOME/goeg/.

В путях к импортируемым пакетам в качестве разделителя элементов пути используется символ «/», даже в Windows. Каждый локальный пакет должен находиться в каталоге с именем, совпадающим с именем пакета. Локальные пакеты могут включать в себя другие локальные пакеты (как, например, path/filepath), подобно пакетам в стандартной библиотеке. (Создание и использование пользовательских пакетов рассматриваются в главе 9.)

Ниже приводится функция main() из простой тестовой программы, вывод которой был продемонстрирован выше:

```
func main() {
    var haystack stack.Stack
    haystack.Push("hay")
    haystack.Push(-15)
    haystack.Push([]string{"pin", "clip", "needle"})
    haystack.Push(81.52)
    for {
        item, err := haystack.Pop()
        if err != nil {
            break
        }
        fmt.Println(item)
    }
}
```

Функция начинается с объявления переменной haystack типа stack.Stack. В соответствии с соглашениями, принятыми в языке Go, ссылки на типы, функции, переменные и другие элементы оформляются в виде пакет.элемент, где пакет – это последний (или единственный) компонент в имени пакета. Это помогает предотвратить конфликты имен. Затем на стек помещается несколько элементов, после чего они выталкиваются со стека и выводятся, пока стек не опустеет.

Одна из замечательных особенностей пользовательского стека состоит в том, что, несмотря на *строгое соблюдение типов* в языке Go, он не ограничен возможностью хранения гомогенных элементов (элементов одного типа), а способен хранить гетерогенные элементы (элементы различных типов). Это достигается благодаря тому,

что тип stack.Stack просто хранит элементы типа interface{} (то есть значения *произвольного* типа) и вообще никак не заботится об их фактических типах. Разумеется, когда программа начинает *использовать* эти значения, их тип становится важным. Однако здесь значения, полученные со стека, просто передаются функции fmt.Println(), которая автоматически, с помощью механизма рефлексии (пакет reflect), определяет типы элементов перед выводом. (Механизм рефлексии будет рассматриваться в последней главе, в §9.4.9.)

Другой замечательной особенностью языка Go, проиллюстрированной в программном коде, является цикл for, не имеющий условного выражения. Это бесконечный цикл, поэтому в большинстве случаев необходимо предусмотреть возможность прерывания такого цикла, например, с помощью инструкции break, как в этом примере, или инструкции return. В следующем примере (§1.6) будет представлен еще один вариант оформления цикла for, а полное описание всех вариантов оформления цикла for приводится в главе 5.

Функции и методы в языке Go могут возвращать одно или несколько значений. В соответствии с соглашениями, принятыми в языке Go, чтобы вернуть признак ошибки, значение ошибки (типа error) возвращается в последнем (или единственном) значении, из числа возвращаемых функций или методом. Пользовательский тип stack.Stack следует этому соглашению.

Теперь, узнав, как используется пользовательский тип stack.Stack, можно перейти к рассмотрению его реализации (в файле stacker/stack/stack.go).

```
package stack
import "errors"
type Stack []interface{}
```

В начале файла, в соответствии с соглашениями, определяется имя пакета. Затем выполняется импортирование других необходимых пакетов – в данном случае импортируется единственный пакет errors.

При определении пользовательского типа происходит связывание идентификатора (имени типа) с новым типом, имеющим такое же базовое представление, как и существующий (встроенный или пользовательский) тип, но который интерпретируется иначе, чем его базовое представление. В данном случае тип Stack – это новое имя среза (то есть ссылка на массив переменной длины) со значениями типа interface{}, и он считается иным типом, нежели []interface{}.

Поскольку все типы в языке Go удовлетворяют требованиям пустого интерфейса, переменная типа `Stack` может хранить значения любого типа.

Все значения встроенных типов коллекций (отображения и срезы), каналы обмена данными (которые могут быть буферизованными) и строки могут возвращать свою длину (или размер буфера) при помощи встроенной функции `len()`. Аналогично срезы и каналы могут сообщать свою емкость (которая может быть больше фактической длины) при помощи встроенной функции `cap()`. (Все встроенные функции языка Go перечислены в табл. 5.1 ниже, где также приводятся ссылки на параграфы с их описанием; срезы рассматриваются в главе 4, в §4.2.) В типах коллекций, пользовательских (наших собственных) и в стандартной библиотеке, обычно предусматривается реализация соответствующих методов `Len()` и `Cap()`, если они имеют смысл.

Поскольку внутренняя реализация типа `Stack` основана на срезе, имеет смысл реализовать в нем методы `Stack.Len()` и `Stack.Cap()`.

```
func (stack Stack) Len() int {
    return len(stack)
}
```

И функции, и методы определяются с помощью ключевого слова `func`. Однако при объявлении методов, после ключевого слова `func` и перед именем метода, указывается заключенный в круглые скобки тип значения, к которому этот метод применяется. После имени функции или метода следует, возможно, пустой, заключенный в круглые скобки список параметров, разделенных запятыми (каждый параметр определяется в форме `имяПеременой тип`). За списком параметров идет открывающая фигурная скобка (если функция или метод ничего не возвращает), или тип возвращаемого значения (такой как тип `int`, возвращаемый методом `Stack.Len()`, представленным выше), или заключенный в круглые скобки список возвращаемых значений, за которым следует открывающая фигурная скобка.

В большинстве случаев вместе с типом указывается имя переменной, к значению которой применяется метод, как показано выше, где использовано имя `stack` (здесь нет конфликта с именем пакета). В терминологии языка Go значение, к которому применяется метод, называется *приемником*[1].

[1] В других языках приемник обычно имеет имя `this` или `self`. В языке Go тоже можно использовать такие имена, но считается, что это не соответствует стилю Go.

В этом примере приемник имеет тип Stack, поэтому приемник передается по значению. Это означает, что любые изменения, произведенные в приемнике, никак не отразятся на оригинальном значении. В таком поведении нет проблемы для методов, не изменяющих приемник, таких как метод Stack.Len(), показанный выше.

Реализация метода Stack.Cap() практически идентична реализации метода Stack.Len() (и потому она не показана здесь). Единственное отличие – в том, что метод Stack.Cap() возвращает результат вызова функции cap(), а не len(). В исходном программном коде имеется также метод Stack.IsEmpty(). Но он тоже похож на метод Stack.Len() – возвращает логическое значение, полученное в результате сравнения с нулем значения, возвращаемого функцией len(), и также не показан здесь.

```
func (stack *Stack) Push(x interface{}) {
    *stack = append(*stack, x)
}
```

Метод Stack.Push() применяется к указателю на значение типа Stack (подробнее об этом чуть ниже) и принимает значение (x) произвольного типа. Встроенная функция append() принимает срез и одно или более значений и возвращает (возможно, новый) срез, включающий содержимое оригинального среза, плюс указанное значение или значения, добавленные в конец (§4.2.3).

Если прежде со стека уже выталкивались элементы (см. реализацию метода Pop() ниже), емкость среза, лежащего в основе стека, наверняка окажется больше его длины, поэтому операция добавления нового элемента на стек будет выполнена очень быстро: новый элемент просто будет записан в срез, в позицию len(stack), а длина стека увеличится на единицу.

Метод Stack.Push() не может совершить ошибку (только если не исчерпает всю память компьютера), поэтому нет необходимости возвращать значение типа error в качестве признака ошибки.

Если метод должен изменять значение приемника, его необходимо определить как указатель[1]. Указатель – это переменная, хранящая адрес в памяти, где находится другое значение. Одна из причин использования указателей – высокая эффективность. Например,

[1] Указатели в языке Go практически ничем не отличаются от указателей в языках C и C++, за исключением того, что Go не поддерживает арифметику с указателями. Подробнее об этом рассказывается в §4.1.

если имеется значение большого типа, намного дешевле передать в параметре указатель на это значение, чем копировать само значение. Другое применение указателей – обеспечение возможности сохранения изменений. Например, когда функции передается переменная (подобно тому, как переменная `stack` передается функции `stack.Len()` в примере выше), она получает копию значения. Это означает, что любые изменения переменной, произведенные внутри функции, никак не отразятся на оригинальном значении. Если функция должна иметь возможность изменить оригинальное значение, как в данном случае, где требуется добавить элемент на стек, оригинальное значение должно передаваться по указателю. В результате внутри функции можно будет изменить значение, на которое ссылается указатель.

Указатель объявляется добавлением символа звездочки (`*`) перед именем типа. Поэтому здесь, в методе `Stack.Push()`, переменная `stack` имеет тип `*Stack`, то есть переменная `stack` хранит указатель на значение типа `Stack`, а не само значение типа `Stack`. Получить доступ к фактическому значению типа `Stack` можно с помощью операции *разыменования* указателя, то есть операции доступа к значению, на которое ссылается указатель. Разыменование выполняется добавлением символа звездочки перед именем переменной. То есть в данном случае обращение к имени `stack` означает обращение к указателю на значение типа `Stack` (получение значения типа `*Stack`), а обращение к имени `*stack` означает разыменование указателя, или обращение к фактическому значению типа `Stack`, на которое ссылается указатель.

Итак, в языке Go (как и в языках C и C++) символ звездочки может означать операцию умножения (когда он находится между двумя числами или переменными, например `x * y`), объявление указателя (когда он предшествует имени типа, например `z *MyType`) и операцию разыменования (когда он предшествует имени переменной-указателя, например `*z`). Не нужно пока проявлять беспокойство по поводу всего вышесказанного: более подробно об указателях в языке Go будет рассказываться в главе 4.

Обратите внимание, что каналы, отображения и срезы в языке Go создаются с помощью функции `make()`, которая всегда возвращает *ссылку* на созданное ею значение. Ссылки действуют практически так же, как указатели, в том смысле, что когда значение передается функции по ссылке, любые изменения, произведенные внутри функции, отражаются на оригинальном канале, отображении или срезе. Однако, в отличие от указателей, ссылки не требуется

разыменовывать, то есть в большинстве случаев нет необходимости добавлять символ звездочки к ним. Но если внутри функции или метода необходимо изменить сам срез, например с помощью функции append() (в противоположность обычной операции изменения значения существующего элемента), тогда его необходимо передавать по указателю или возвращать новый срез (и присваивать значение, возвращаемое функцией или методом, оригинальной переменной), потому что иногда функция append() возвращает другой срез, отличный от полученного ею.

Для представления типа Stack используется срез, поэтому значения типа Stack можно передавать функциям, выполняющим операции со срезами, таким как append() и len().

Тем не менее значения типа Stack – это совсем другие значения, отличающиеся от представления, лежащего в их основе, поэтому при необходимости изменения этих значений их следует передавать по указателю.

```
func (stack Stack) Top() (interface{}, error) {
    if len(stack) == 0 {
        return nil, errors.New("can't Top() an empty stack")
    }
    return stack[len(stack)-1], nil
}
```

Метод Stack.Top() возвращает элемент, находящийся на вершине стека (то есть элемент, добавленный последним), и пустое значение nil в качестве ошибки или пустое значение nil в качестве элемента и непустое значение ошибки, если стек пуст. Приемник в данном случае передается по значению, поскольку стек не изменяется.

Возвращаемое значение error имеет тип interface (§6.3), который определяет единственный метод, Error()string. В общем случае библиотечные функции в языке Go возвращают значение error последним (или единственным) в списке возвращаемых значений, чтобы обозначить успешное выполнение (когда error получает значение nil) или неудачу. В этом примере тип Stack действует, подобно типам из стандартной библиотеки, создавая новое значение error с помощью функции errors.New() из пакета errors.

Значение nil в языке Go используется как значение пустых указателей (и пустых ссылок), то есть указателей, никуда не указывающих,

и ссылок, ни на что не ссылающихся[1]. Такие указатели должны использоваться лишь в операциях сравнения или присваивания, методы к ним обычно не применяются.

Конструкторы в языке Go никогда не вызываются неявно. Вместо этого Go гарантирует, что при создании значения оно обязательно будет инициализировано нулевым значением. Например, числовые переменные инициализируются значением 0, строковые – пустыми строками, указатели – значением `nil`. Поля структур инициализируются аналогично переменным. Поэтому в языке Go не может быть неинициализированных переменных, что устраняет основной источник ошибок, приносящих немало расстройств в других языках программирования. Если по каким-то причинам нулевое значение не устраивает, можно написать функцию-конструктор и вызывать ее явно, как это делалось выше для создания нового значения `error`. Имеется также возможность предотвратить создание значений типа, минуя вызов функции-конструктора, как будет показано в главе 6.

Если стек непустой, метод возвращает значение, находящееся на вершине стека и значение `nil` в качестве признака ошибки. Поскольку в языке Go отсчет элементов срезов и массивов начинается с нуля, первый элемент находится в позиции 0, а последний – в позиции `len(срезИлиМассив) - 1`.

При возврате из функции или метода более одного значения не предъявляется никаких формальных требований. Достаточно просто перечислить типы возвращаемых значений после имени функции или метода и предусмотреть хотя бы одну инструкцию `return` с соответствующим списком возвращаемых значений.

```
func (stack *Stack) Pop() (interface{}, error) {
    theStack := *stack
    if len(theStack) == 0 {
        return nil, errors.New("can't Pop() an empty stack")
    }
    x := theStack[len(theStack)-1] ❶
    *stack = theStack[:len(theStack)-1] ❷
    return x, nil
}
```

[1] Значение `nil` в языке Go играет почти ту же роль, что и значение `NULL` или `0` в C и C++, а также значение `null` в Java и `nil` в Objective-C.

Метод `Stack.Pop()` удаляет значение с вершины стека (добавленное последним) и возвращает его. Подобно методу `Stack.Top()`, он возвращает элемент стека и значение `nil` в качестве ошибки или, если стек пуст, значение `nil` в качестве элемента и непустое значение ошибки.

Приемник должен передаваться методу в виде указателя, потому что он изменяет стек, удаляя возвращаемый элемент. Для удобства, чтобы внутри метода не ссылаться на приемник как `*stack` (переменная `stack` указывает на стек), фактический стек присваивается локальной переменной (`theStack`), и затем все операции выполняются с этой переменной. Это довольно выгодно, потому что ссылка `*stack` указывает на значение типа `Stack`, внутреннее представление которого основано на срезе, то есть в действительности здесь присваивается чуть больше, чем просто ссылка на срез.

Если стек пуст, возвращается соответствующее значение ошибки. В противном случае с вершины стека снимается (последний добавленный) элемент и сохраняется в локальной переменной (`x`). Затем извлекается фрагмент массива (который сам является срезом). Новый срез, содержащий на один элемент меньше, чем оригинальный срез, немедленно устанавливается в качестве значения, на которое ссылается указатель `stack`. В конце метод возвращает извлеченное со стека значение и `nil` в качестве ошибки. Вполне оправданно было бы ожидать, что любой приличный компилятор Go будет повторно использовать тот же срез, просто уменьшая его длину на единицу, оставляя неизменной его емкость, вместо копирования всех данных в новый срез.

Возвращаемый элемент извлекается оператором индексирования `[]` с единственным индексом (❶) – в данном случае с индексом последнего элемента в срезе.

Новый срез получается оператором извлечения среза `[]` с диапазоном индексов (❷). Диапазон индексов указывается в формате `первый:последний`. Если `первый` индекс опущен, как в данном случае, вместо него используется значение 0, а если опущен `последний` индекс, вместо него используется значение вызова функции `len()` для среза. Полученный таким способом срез будет содержать элементы с индексами, начиная с `первого` *включительно* и до `последнего`, *исключая* его. То есть, если указать индекс последнего элемента на единицу меньше длины среза, будет получен срез от первого до последнего (не включая его) элемента, что равносильно удалению последнего элемента из среза. (Индексирование срезов рассматривается в главе 4, в §4.2.1.)

В данном примере в качестве приемников используются значения типа `Stack`, а не указатели (то есть значения типа `*Stack`) в тех методах, где не требуется изменять стек. Для пользовательских типов с легковесным внутренним представлением (например, построенных на основе нескольких значений типа `int` или `string`) это вполне разумно. Но для тяжелых пользовательских типов обычно намного эффективнее в качестве *приемников* передавать *указатели*, потому что операция передачи указателя (который обычно является простым 32- или 64-битным значением) намного дешевле, чем передача объемного значения, даже в методах, не изменяющих значения приемника.

Важно также отметить, что если метод применяется к значению, но требует передать ему указатель на значение, компилятор Go поймет, что методу нужно передать адрес значения (то есть адресуемое значение – см. §6.2.1), а не его копию. Соответственно, если метод применяется к указателю на значение, но требует передать ему само значение, компилятор Go поймет, что нужно разыменовать указатель и передать методу значение, на которое он указывает[1].

Как демонстрирует этот пример, создание пользовательских типов в языке Go выполняется достаточно просто и не отягощено излишними формальностями, характерными для многих других языков. Подробно об объектно-ориентированных особенностях языка Go рассказывается в главе 6.

1.6. Американизация – файлы, отображения и замыкания

Чтобы иметь хоть какое-то практическое применение, язык программирования должен обеспечивать инструменты для чтения и записи внешних данных. В предыдущих разделах мы мельком увидели несколько функций вывода из пакета `fmt`. В этом разделе будут представлены основные инструменты для работы с файлами, имеющиеся в языке Go. Здесь также будут рассматриваться некоторые дополнительные особенности языка, такие как интерпретация функций и методов как обычных значений, что делает возможным передавать их в виде параметров. Кроме того, здесь будет задействован тип `map` (также известный как словарь или хеш).

[1] Именно поэтому в языке Go отсутствует оператор `->`, используемый в языках C и C++.

В этом разделе будет представлено достаточно сведений, необходимых для создания программ, выполняющих операции ввода/вывода с текстовыми файлами, что сделает примеры и упражнения более интересными. Подробнее об инструментах для работы с файлами рассказывается в главе 8.

Примерно в середине XX века американский английский превзошел британский английский по распространенности. В примере, демонстрируемом в этом разделе, будет представлена программа, которая читает данные из текстового файла и копирует содержимое этого файла в новый файл, замещая при этом слова из британского английского языка их эквивалентами из американского английского. (Разумеется, программа при этом не учитывает различий в семантике и не распознает идиоматических выражений.) Исходный текст программы хранится в файле `americanise/americanise.go` и будет рассматриваться сверху вниз, начиная с раздела импортирования пакетов, затем мы разберем функцию `main()`, потом функции, вызываемые функцией `main()`, и т. д.

```
import (
    "bufio"
    "fmt"
    "io"
    "io/ioutil"
    "log"
    "os"
    "path/filepath"
    "regexp"
    "strings"
)
```

Все пакеты, импортируемые программой `americanise`, входят в состав стандартной библиотеки Go. Пакеты могут вкладываться друг в друга без лишних формальностей, подобно пакету `ioutil`, включенному в пакет `io`, и пакету `filepath`, включенному в пакет `path`.

Пакет `bufio` предоставляет функции буферизованного ввода/вывода, включая функции чтения и записи строк из и в текстовые файлы в кодировке UTF-8. Пакет `io` предоставляет низкоуровневые функции ввода/вывода, а также интерфейсы `io.Reader` и `io.Writer`, необходимые функции `americanise()`. Пакет `io/ioutil` – высокоуровневые функции для работы с файлами. Пакет `regexp` дает поддержку

регулярных выражений. Другие пакеты (fmt, log, filepath и strings) уже упоминались в предыдущих разделах.

```
func main() {
    inFilename, outFilename, err := filenamesFromCommandLine() ❶
    if err != nil {
        fmt.Println(err) ❷
        os.Exit(1)
    }
    inFile, outFile := os.Stdin, os.Stdout ❸
    if inFilename != "" {
        if inFile, err = os.Open(inFilename); err != nil {
            log.Fatal(err)
        }
        defer inFile.Close() ❹
    }
    if outFilename != "" {
        if outFile, err = os.Create(outFilename); err != nil {
            log.Fatal(err)
        }
        defer outFile.Close() ❺
    }
    if err = americanise(inFile, outFile); err != nil {
        log.Fatal(err)
    }
}
```

Функция main() получает из аргументов командной строки имена входного и выходного файлов, создает соответствующие значения файлов и затем передает файлы функции americanise() для обработки.

Функция начинается с извлечения имен файлов для чтения и записи и значения error. Если в процессе разбора аргументов командной строки возникла ошибка, выводится сообщение (содержащее описание порядка использования программы) и программа завершается. Некоторые функции вывода в языке Go используют механизм рефлексии (интроспекции) для вывода значения с помощью его метода Error() string, если имеется, или метода String() string, если имеется, или другого метода, который может предложить это значение. Если предусмотреть в реализации своих собственных типов один из этих методов, функции вывода автоматически смогут выводить значения пользовательских типов, как будет показано в главе 6.

Если переменная `err` имеет значение `nil`, значит, переменные `inFilename` и `outFilename` хранят строки (могут быть пустыми строками) и можно продолжать. Файлы в языке Go представлены указателями на значения типа `os.File`, потому в программе создаются две переменные этого типа и инициализируются ссылками на потоки стандартного ввода и вывода (оба имеют тип `*os.File`). Так как функции и методы в языке Go могут возвращать несколько значений, из этого следует, что Go поддерживает множественное присваивание, как в данном примере (❶, ❸ в листинге выше).

Оба имени файлов обрабатываются практически одинаково. Если именем файла является пустая строка, считается, что в качестве файла уже используется значение `os.Stdin` или `os.Stdout` (каждое из которых имеет тип `*os.File`, то есть является указателем на значение типа `os.File`, представляющее файл). Но если имя файла – не пустая строка, создается новое значение `*os.File` для чтения или записи в соответствующий файл.

Функция `os.Open()` принимает имя файла и возвращает значение типа `*os.File`, которое можно использовать для чтения файла. Аналогично функция `os.Create()` принимает имя файла и возвращает значение типа `*os.File`, которое можно использовать для чтения из файла или записи в файл, создавая файл, если он не существует, и усекая его размер до нуля, если существует. (В языке Go имеется также функция `os.OpenFile()`, позволяющая управлять флагами режимов и разрешений, используемых при открытии файла.)

В действительности функции `os.Open()`, `os.Create()` и `os.OpenFile()` возвращают два значения: `*os.File` и `nil`, если файл был открыт успешно, или `nil` и `error` в случае ошибки.

Если переменная `err` имеет значение `nil`, известно, что файл был открыт успешно, поэтому можно сразу выполнить инструкцию `defer`, чтобы закрыть файл. Любая функция, передаваемая инструкции `defer` (§5.5), должна вызываться, поэтому после имени функции указываются круглые скобки (❹ и ❺ в листинге выше), но фактический вызов происходит в момент завершения функции, вызвавшей инструкцию `defer`. То есть инструкция `defer` «замораживает» вызов функции, откладывая его на более позднее время. Это означает, что на выполнение инструкции `defer` почти не расходуется времени и управление немедленно передается следующей за ней инструкции. Таким образом, задержанный вызов метода `os.File.Close()` не будет выполнен, пока не завершится вызывающая функция (или не возникнет *аварийная ситуация*, о чем рассказывается чуть ниже) – в данном случае функция `main()`, поэтому

файл остается открытым на протяжении всего времени работы с ним и гарантированно закрывается по завершении или в случае аварии.

Если попытка открыть файл завершается ошибкой, вызывается функция `log.Fatal()`, которой передается значение ошибки. Как уже отмечалось в предыдущем разделе, эта функция выводит дату, время и ошибку (в поток `os.Stderr`, если явно не определен другой файл для вывода сообщений) и завершает программу вызовом `os.Exit()`. Когда вызывается функция `os.Exit()` (напрямую или посредством функции `log.Fatal()`), программа немедленно завершает работу, и все отложенные вызовы теряются. Однако в этом нет никакой проблемы, потому что система времени выполнения языка Go автоматически закроет все открытые файлы, механизм сборки мусора освободит память, занятую программой, и корректно будут закрыты все сетевые соединения и подключения к базам данных, с которыми программа могла обмениваться данными. Как и в примере `bigdigits`, здесь не используется функция `log.Fatal()` в первой инструкции `if` (❷ в листинге выше), потому что переменная `err` содержит текст с описанием порядка использования программы, который должен выводиться без указания даты и времени, которые выводит функция `log.Fatal()`.

Аварией в языке Go называется ошибка времени выполнения (напоминает исключения в других языках). Аварийную ситуацию можно создать вручную, вызвав встроенную функцию `panic()`, а остановить развитие аварии можно с помощью функции `recover()` (§5.5). Теоретически функции `panic/recover` можно использовать для реализации универсального механизма исключений, но это считается плохой практикой. В языке Go принято сообщать об ошибках, возвращая значение ошибки из функций и методов в виде единственного или последнего возвращаемого значения или `nil`, при отсутствии ошибки, и проверять наличие ошибки в вызывающем программном коде. Функции `panic/recover` предназначены для обработки по-настоящему исключительных (то есть неожиданных) ситуаций, а не обычных ошибок[1].

В случае успешного открытия обоих файлов (файлы `os.Stdin`, `os.Stdout` и `os.Stderr` открываются системой времени выполнения

[1] Подход, принятый в языке Go, существенно отличается от подхода в языках C++, Java и Python, где часто механизм исключений используется и для обработки исключительных ситуаций, и для обработки ошибок. Обсуждение и разъяснение действия механизма `panic/recover` в языке Go можно найти по адресу: https://groups.google.com/group/golang-nuts/browse_thread/thread/1ce5cd050bb973e4?pli=1.

языка Go автоматически) можно вызвать функцию americanise() и передать ей файлы для обработки. Если функция americanise() вернет значение nil, функция main() завершит работу нормальным образом и выполнятся все отложенные вызовы – в данном случае вызовы функций, закрывающих файлы inFile и outFile, если они не являются потоками os.Stdin и os.Stdout. В противном случае, если переменная err будет иметь значение, отличное от nil, будет выведено сообщение об ошибке, программа завершится, и все открытые файлы будут закрыты системой времени выполнения языка Go.

Функция americanise() принимает значения типа io.Reader и io.Writer, а не *os.File, но это не имеет большого значения, потому что тип os.File поддерживает интерфейс io.ReadWriter (который является простым объединением интерфейсов io.Reader и io.Writer) и, следовательно, может использоваться везде, где ожидается получить значение типа io.Reader или io.Writer. Это пример динамической (утиной) типизации в действии – параметрами функции americanise() являются интерфейсы, поэтому функция может принимать любые значения, независимо от их типов, при условии что они реализуют ожидаемые интерфейсы, то есть любые значения, имеющие методы, определяемые интерфейсами. В случае успеха функция americanise() возвращает значение nil и ошибку – в случае неудачи.

```
func filenamesFromCommandLine() (inFilename, outFilename string,
    err error) {
    if len(os.Args) > 1 && (os.Args[1] == "-h" || os.Args[1] == "--help") {
        err = fmt.Errorf("usage: %s [<]infile.txt [>]outfile.txt",
            filepath.Base(os.Args[0]))
        return "", "", err
    }
    if len(os.Args) > 1 {
        inFilename = os.Args[1]
        if len(os.Args) > 2 {
            outFilename = os.Args[2]
        }
    }
    if inFilename != "" && inFilename == outFilename {
        log.Fatal("won't overwrite the infile")
    }
    return inFilename, outFilename, nil
}
```

Функция `filenamesFromCommandLine()` возвращает две строки и значение ошибки, и, в отличие от представленных выше функций, здесь для возвращаемых значений определяются не только их типы, но и имена. Благодаря этому при входе в функцию возвращаемым переменным присваиваются *пустые значения* (в данном случае пустые строки и значение `nil`), которые остаются такими, если переменным явно не будут присвоены другие значения в теле функции. (Подробнее эта тема будет рассматриваться ниже, при обсуждении функции `americanise()`.)

Начинается функция с проверки, не запросил ли пользователь справку о порядке использования программы[1]. В этом случае создается новое значение `error` вызовом функции `fmt.Errorf()`, которой передается строка с описанием порядка использования программы и управление возвращается вызывающей программе. Как это принято в языке Go, вызывающая программа должна проверить возвращаемое значение ошибки и выполнить необходимые операции (именно так и действует функция `main()`). Функция `fmt.Errorf()` похожа на уже знакомую функцию `fmt.Printf()`, за исключением того, что она не выводит в `os.Stdout` значение ошибки, содержащее строку, сформированную с помощью указанной строки формата и аргументов, а возвращает его. (Для создания значения ошибки из строкового литерала используется функция `errors.New()`.)

Если пользователь не запросил справочную информацию, далее функция проверяет наличие каких-либо аргументов командной строки и присваивает первый аргумент возвращаемой переменной `inFilename`, а второй аргумент – возвращаемой переменной `outFilename`. Разумеется, пользователь может не передать ни одного аргумента командной строки, и в этом случае обе переменные, `inFilename` и `outFilename`, останутся пустыми строками; или он может указать только один аргумент, тогда переменная `inFilename` получит значение аргумента, а переменная `outFilename` останется пустой.

В конце выполняется простая проверка, чтобы предотвратить затирание входного файла выходными данными, завершающая программу при необходимости. Если проверка прошла успешно,

[1] Стандартная библиотека языка Go включает пакет `flag` для обработки аргументов командной строки. На сайте godashboard.appspot.com/project доступны сторонние пакеты для обработки аргументов командной строки способом, совместимым с GNU. (Особенности использования сторонних пакетов описываются в главе 9.)

функция возвращает управление[1]. Функции и методы, возвращающие одно значение или более, *должны* иметь хотя бы одну инструкцию `return`. Добавление имен к возвращаемым типам, как в данной функции, повышает удобочитаемость программного кода и может пригодиться для создания документации с помощью инструмента `godoc`. Если в функции или методе, помимо типов, указаны также имена возвращаемых переменных, допускается использовать пустую инструкцию `return` (то есть без указания значений или переменных). В таких случаях возвращаются переменные, перечисленные в заголовке функции. В этой книге не используются пустые инструкции `return`, потому что это не соответствует хорошему стилю программирования на языке Go.

В языке Go реализован непротиворечивый подход к чтению и записи данных, позволяющий выполнять операции чтения и записи с файлами, буферами (например, срезами байт или строками), со стандартными потоками ввода, вывода и ошибок, а также со значениями пользовательских типов, при условии что они реализуют методы, определяемые интерфейсами чтения и записи.

Чтобы значение было доступно для стандартных операций чтения, оно должно удовлетворять требованиям интерфейса `io.Reader`. Этот интерфейс определяет единственный метод с сигнатурой `Read([]byte) (int, error)`. Метод `Read()` читает данные из значения, относительно которого он вызывается, и помещает прочитанные данные в указанный срез с байтами. Он должен возвращать: количество прочитанных байтов и значение ошибки или `nil` при ее отсутствии; или `io.EOF` (признак «конца файла»), в случае исчерпания данных и отсутствия ошибки или какое-то другое значение, отличное от `nil`, в случае ошибки. Аналогично, чтобы значение было доступно для стандартных операций записи, оно должно удовлетворять требованиям интерфейса `io.Writer`. Этот интерфейс определяет единственный метод с сигнатурой `Write([]byte) (int, error)`. Метод `Write()` должен записывать данные из указанного среза с байтами в значение, относительно которого он вызывается, и должен возвращать количество записанных байтов и значение ошибки (которое может быть значением `nil` при отсутствии ошибки).

[1] В действительности у пользователя имеется возможность затереть исходный файл, воспользовавшись механизмом перенаправления, например `$ ./americanise infile > infile`, но здесь предотвращается хотя бы самый очевидный случай.

Пакет io предоставляет инструменты для чтения и записи, но они не используют буферизацию и оперируют в терминах простых байтов. Пакет bufio предоставляет инструменты ввода/вывода с буферизацией, где инструменты ввода могут работать только со значениями, реализующими интерфейс io.Reader (то есть предоставляющими соответствующий метод Read()), а инструменты вывода – только со значениями, реализующими интерфейс io.Writer (то есть предоставляющими соответствующий метод Write()). Инструменты чтения и записи из пакета bufio обеспечивают возможность буферизации, могут оперировать в терминах байтов и строк и потому лучше подходят для чтения и записи текстовых файлов в кодировке UTF-8.

```
var britishAmerican = "british-american.txt"
func americanise(inFile io.Reader, outFile io.Writer) (err error) {
    reader := bufio.NewReader(inFile)
    writer := bufio.NewWriter(outFile)
    defer func() {
        if err == nil {
            err = writer.Flush()
        }
    }()
    var replacer func(string) string ❶
    if replacer, err = makeReplacerFunction(britishAmerican); err != nil {
        return err
    }
    wordRx := regexp.MustCompile("[A-Za-z]+")
    eof := false
    for !eof {
        var line string ❷
        line, err = reader.ReadString('\n')
        if err == io.EOF {
            err = nil  // в действительности признак io.EOF не является ошибкой
            eof = true // это вызовет прекращение цикла в следующей итерации
        } else if err != nil {
            return err // в случае настоящей ошибки выйти немедленно
        }
        line = wordRx.ReplaceAllStringFunc(line, replacer)
        if _, err = writer.WriteString(line); err != nil { ❸
            return err
        }
    }
    return nil
}
```

Функция `americanise()` добавляет буферизацию, создавая значения для чтения и записи файлов `inFile` и `outFile`. Затем читает строки и записывает каждую из них, замещая слова, характерные для британского английского, их американскими эквивалентами.

Функция начинается с создания значений для буферизованного ввода/вывода, посредством которых можно обращаться к содержимому как к двоичным байтам или, что более удобно в данном случае, как к строкам. Функция-конструктор `bufio.NewReader()` принимает в виде аргумента любое значение, поддерживающее интерфейс `io.Reader` (то есть любое значение, имеющее метод `Read()`), и возвращает новое значение буферизованного ввода типа `io.Reader`, которое будет читать данные из указанного значения. Аналогично действует функция `bufio.NewWriter()`. Обратите внимание, что функция `americanise()` ничего не знает, откуда выполняется чтение и куда производится запись, – значения для чтения и записи могут представлять сжатые файлы, сетевые соединения, срезы с байтами (`[]byte`) и все, что угодно, что поддерживает интерфейсы `io.Reader` и `io.Writer`. Такой способ организации операций с помощью интерфейсов обеспечивает высокую гибкость и простоту реализации функциональных возможностей на языке Go.

Далее создается анонимная отложенная функция, которая выталкивает содержимое буфера записи, перед тем как функция `americanise()` вернет управление вызывающей программе. Анонимная функция будет вызвана при нормальном завершении функции `americanise()` или в случае аварии. Если в процессе выполнения никаких ошибок не возникло и в буфере остались незаписанные байты, они будут записаны до того, как функция `americanise()` вернет управление. Поскольку при выталкивании данных из буфера может произойти ошибка, переменной `err` присваивается значение, возвращаемое методом `writer.Flush()`. Для реализации менее надежного варианта достаточно было бы простой инструкции `defer writer.Flush()`, гарантирующей, что значение для записи вытолкнет буфер до выхода из вызывающей функции, и игнорирующей любые ошибки, которые могут произойти в процессе выталкивания буфера.

В языке Go допускается использовать именованные возвращаемые значения, и здесь так же используются преимущества этой возможности (`err error`), как это было сделано в функции `filenamesFromCommandLine()`, выше. Однако имейте в виду, что правила видимости именованных возвращаемых значений имеют свои особенности, о чем следует помнить при их использовании. Например, если имеется именованное