

Программирование на языке **Go**



Марк Саммерфильд

Марк Саммерфильд

Программирование на Go

Разработка приложений XXI века



Москва, 2013

Mark Summerfield

Programming in Go:

Creating Applications for the 21st Century



Addison-Wesley

Марк Саммерфильд

Программирование на Go

Разработка приложений XXI века



Москва, 2013

УДК 004.438Go
ББК 32.973.26-018.1
C17

Марк Саммерфильд

C17 Программирование на Go. Разработка приложений XXI века:
пер. с англ.: Киселёв А. Н. – М.: ДМК Пресс, 2013. – 580 с.: ил.
ISBN 978-5-94074-854-0

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.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-0-321-77463-7 (англ.)
ISBN 978-5-94074-854-0 (рус.)

© Copyright 2012 Qtrac Ltd.
© Оформление, ДМК Пресс, 2013



Содержание

Введение	11
Зачем изучать язык Go?	12
Структура книги	16
Благодарности	17
 1. Обзор в пяти примерах.....	19
1.1. Начало	19
1.2. Правка, компиляция и запуск	22
1.3. Hello кто?.....	28
1.4. Большие цифры – двумерные срезы	32
1.5. Стек – пользовательские типы данных с методами.....	38
1.6. Американизация – файлы, отображения и замыкания	49
1.7. Из полярных координат в декартовы – параллельное программирование	65
1.8. Упражнение.....	74
 2. Логические значения и числа	76
2.1. Начальные сведения	76
2.1.1. Константы и переменные	78
2.2. Логические значения и выражения.....	83
2.3. Числовые типы	84
2.3.1. Целочисленные типы	87
2.3.2. Вещественные типы.....	93

2.4. Пример: statistics.....	103
2.4.1. Реализация простых статистических функций	104
2.4.2. Реализация простого HTTP-сервера	106
2.5. Упражнения.....	111
3. Строки.....	113
3.1. Литералы, операторы и экранированные последовательности	115
3.2. Сравнение строк	117
3.3. Символы и строки	121
3.4. Индексирование и получение срезов строк.....	124
3.5. Форматирование строк с помощью пакета fmt	128
3.5.1. Форматирование логических значений	134
3.5.2. Форматирование целочисленных значений	134
3.5.3. Форматирование символов	136
3.5.4. Форматирование вещественных значений.....	137
3.5.5. Форматирование строк и срезов	139
3.5.6. Форматирование для отладки.....	141
3.6. Другие пакеты для работы со строками	145
3.6.1. Пакет strings	145
3.6.2. Пакет strconv.....	153
3.6.3. Пакет utf8.....	158
3.6.4. Пакет unicode.....	160
3.6.5. Пакет regexr	161
3.7. Пример: m3u2pls.....	173
3.8. Упражнения.....	180
4. Типы коллекций	183
4.1. Значения, указатели и ссылочные типы	184
4.2. Массивы и срезы.....	195
4.2.1. Индексирование срезов и извлечение срезов из срезов.....	201

4.2.2. Итерации по срезам	202
4.2.3. Изменение срезов	204
4.2.4. Сортировка и поиск по срезам.....	209
4.3. Отображения	214
4.3.1. Создание и заполнение отображений	216
4.3.2. Поиск в отображениях	219
4.3.3. Изменение отображений	220
4.3.4. Итерации по отображениям с упорядоченными ключами	221
4.3.5. Инвертирование отображений.....	222
4.4. Примеры	223
4.4.1. Пример: угадай разделитель	223
4.4.2. Пример: частота встречаемости слов	226
4.5. Упражнения.....	234
5. Процедурное программирование.....	238
5.1. Введение в инструкции	238
5.1.1. Преобразование типа	243
5.1.2. Приведение типов	245
5.2. Ветвление	247
5.2.1. Инструкция if	247
5.2.2. Инструкция switch	249
5.3. Инструкция цикла for	259
5.4. Инструкции организации взаимодействия и параллельного выполнения.....	263
5.4.1. Инструкция select	267
5.5. Инструкция defer и функции panic() и recover()	272
5.5.1. Функции panic() и recover()	273
5.6. Пользовательские функции	281
5.6.1. Аргументы функций	283
5.6.2. Функции init() и main()	287
5.6.3. Замыкания.....	289
5.6.4. Рекурсивные функции.....	291

5.6.5. Выбор функции во время выполнения.....	295
5.6.6. Обобщенные функции.....	298
5.6.7. Функции высшего порядка.....	305
5.7. Пример: сортировка с учетом отступов	312
5.8. Упражнения.....	319
6. Объектно-ориентированное программирование ..	322
6.1. Ключевые понятия.....	323
6.2. Пользовательские типы.....	326
6.2.1. Добавление методов	328
6.2.2. Типы с проверкой.....	334
6.3. Интерфейсы.....	336
6.3.1. Встраивание интерфейсов	343
6.4. Структуры	348
6.4.1. Структуры: агрегирование и встраивание	349
6.5. Примеры	357
6.5.1. Пример: FuzzyBool – пользовательский тип с единственным значением	357
6.5.2. Пример: фигуры – семейство пользовательских типов.....	365
6.5.3. Пример: упорядоченное отображение – обобщенный тип коллекций	381
6.6. Упражнения.....	392
7. Параллельное программирование	397
7.1. Ключевые понятия.....	399
7.2. Примеры	406
7.2.1. Пример: фильтр	407
7.2.2. Пример: параллельный поиск	412
7.2.3. Пример: поточно-ориентированное отображение	422
7.2.4. Пример: отчет о работе веб-сервера	430
7.2.5. Пример: поиск дубликатов.....	441
7.3. Упражнения.....	451

8. Обработка файлов..... 455

8.1. Файлы с пользовательскими данными	455
8.1.1. Обработка файлов в формате JSON.....	460
8.1.2. Обработка файлов в формате XML.....	467
8.1.3. Обработка простых текстовых файлов	475
8.1.4. Обработка файлов в двоичном формате Go	484
8.1.5. Обработка файлов в пользовательском двоичном формате.....	488
8.2. Архивные файлы	499
8.2.1. Создание zip-архивов	499
8.2.2. Создание тарболлов	502
8.2.3. Распаковывание zip-архивов	504
8.2.4. Распаковывание тарболлов	506
8.3. Упражнения.....	509

9. Пакеты 512

9.1. Пользовательские пакеты	512
9.1.1. Создание пользовательских пакетов	513
9.1.2. Импортирование пакетов	523
9.2. Сторонние пакеты	524
9.3. Краткий обзор команд компилятора Go	525
9.4. Краткий обзор стандартной библиотеки языка Go	526
9.4.1. Пакеты для работы с архивами и сжатыми файлами	527
9.4.2. Пакеты для работы с байтами и строками	527
9.4.3. Пакеты для работы с коллекциями	529
9.4.4. Пакеты для работы с файлами и ресурсами операционной системы.....	532
9.4.5. Пакеты для работы с графикой	534
9.4.6. Математические пакеты	534
9.4.7. Различные пакеты.....	535
9.4.8. Пакеты для работы с сетью	536
9.4.9. Пакет reflect	537
9.5. Упражнения.....	541



А. Эпилог	545
В. Опасность патентов на программное обеспечение	548
С. Список литературы	553
Предметный указатель	556

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 наличием 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 легкие 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 С 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 С++, Java или Python. Она намного проще в использовании и значительно гибче.

Как и в языке С, в Go отсутствуют генерики (generics) (шаблоны, в терминологии С++), 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 С и С++. А благодаря отсутствию препроцессора семантика программы не может измениться незаметно для программиста, как это может произойти при небрежном обращении с директивами `#define` в языках С и С++.

Возможно, языки С++, Objective-C и Java создавались как улучшенные версии языка С (а последний – как улучшенная версия языка С++). В языке Go тоже можно усмотреть попытку создать улучшенную версию языка С, 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 С, чем к любым другим языкам, и его можно считать попыткой устранить недостатки языка С, 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 Манчини (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 В.

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 экономить 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, сообщаящий, 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-8¹. 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 компоновка¹. 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 отладки¹.

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 присваиваемому 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	7777	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] + " " ❹
    } 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 Go¹. 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 (❶ 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 (❷ 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` (❸ 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 `имяПеременной 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 указатель¹. 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 требуются

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 ссылающихся¹. 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 указывает¹.

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, ❷ 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 (❸ и ❹ 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 ошибок¹.

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 программы¹. 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 управление¹. 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

возвращаемое значение `value`, ему можно присваивать значения в любой точке внутри функции с помощью оператора присваивания (`=`). Однако, если внутри инструкции `if` имеется такая инструкция, как `value := ...`, будет создана новая переменная, потому что инструкция `if` образует новый блок. В результате переменная `value` внутри блока инструкции `if` как бы загородит собой возвращаемое значение `value`. В функции `americanise()` имя `err` соответствует возвращаемому значению, поэтому необходимо гарантировать, что этому имени нигде в данной функции не будет присваиваться какое-либо значение с помощью оператора сокращенного объявления переменной (`:=`), чтобы избежать риска создания теневой переменной. Вследствие вышесказанного приходится явно объявлять переменные, которым неоднократно будут присваиваться значения в течение всего времени выполнения функции, как в случае с переменной `replacer` (❶ в листинге выше), хранящей ссылку на функцию, и переменной `line` (❷ в листинге выше), куда выполняется чтение данных. Альтернативное решение заключается в том, чтобы отказаться от именованных возвращаемых значений и возвращать все необходимые значения явно.

Обратите также внимание на использование *пустого идентификатора*, `_` (❸ в листинге выше). Пустой идентификатор играет роль заполнителя в операции присваивания, где ожидается переменная, и помогает просто отбросить присваиваемое значение. Пустой идентификатор не считается новой переменной, поэтому при использовании с оператором `:=`, будет создана как минимум одна, другая (новая) переменная.

Стандартная библиотека Go содержит мощный пакет `regexp` (§3.6.5) поддержки регулярных выражений. Этот пакет можно использовать для создания указателей на значения типа `regexp.Regexp` (то есть для создания значений типа `*regexp.Regexp`). Эти значения обладают множеством методов поиска и замены. В данной функции используется метод `regexp.Regexp.ReplaceAllStringFunc()`, который принимает строку и функцию с сигнатурой `func(string) string`, вызывает эту функцию для каждого найденного совпадения, передает ей совпавший текст и замещает совпадение текстом, возвращаемым функцией.

Если бы функция, возвращающая замещающий текст, была очень маленькой, например как простое преобразование в верхний регистр символов в совпавших словах, ее можно было бы реализовать в виде анонимной функции, как показано ниже:

```
line = wordRx.ReplaceAllStringFunc(line,  
    func(word string) string { return strings.ToUpper(word) })
```

Однако, хотя функция определения замещающего текста в программе `americanise` содержит всего несколько строк, тем не менее она требует выполнения некоторых подготовительных операций, поэтому была создана другая функция, `makeReplacerFunction()`, которой передается имя файла, содержащего строки с оригинальными и замещающими их словами. Она возвращает функцию, отыскивающую соответствующие замены.

Если от функции `makeReplacerFunction()` будет получено непустое значение ошибки, она возвращается вызывающей программе, которая должна проверить ее и отреагировать соответственно.

Регулярные выражения могут быть скомпилированы с помощью функции `regexp.Compile()`, возвращающей значения `*regexp.Regexp` и `nil` или `nil` и значение ошибки, если регулярное выражение оказалось недопустимым. Этот способ идеально подходит для случая, когда регулярные выражения поступают из внешних источников, например извлекаются из файлов или вводятся пользователем. В данном же примере использована функция `regexp.MustCompile()` — она просто возвращает `*regexp.Regexp` или возбуждает аварийную ситуацию, если регулярное выражение содержит ошибки. Регулярному выражению, использованному в этом примере, соответствует самая длинная последовательность из алфавитных символов латиницы.

За инструкциями подготовки функции, реализующей замену, и регулярного выражения следует бесконечный цикл, который начинается с чтения строки. Метод `bufio.Reader.ReadString()` читает (или, строго говоря, *декодирует*) последовательность двоичных байтов, как текст в кодировке UTF-8 (этот метод также может декодировать текст в 7-битной кодировке ASCII), до байта с указанным значением (включая его) или до конца файла. Вызывающей программе возвращается прочитанный текст в виде значения типа `string` и значение ошибки (или `nil`).

Значение ошибки, отличное от `nil`, метод `bufio.Reader.ReadString()` может вернуть либо при достижении конца входных данных, либо в случае действительной ошибки. В случае достижения конца входных данных переменная `err` получит значение `io.EOF`, что в действительности не является ошибкой, поэтому переменной `err` присваивается значение `nil`, а переменной `eof` — значение `true`, благодаря чему обеспечивается завершение цикла в следующей итерации и предотвращается попытка чтения данных за пределами файла. Получив признак `io.EOF`, не следует тут же завершать цикл, потому что вполне возможно, что последняя строка в файле просто не заканчивается символом перевода строки, и ее необходимо обработать как обычно, в дополнение к обработке ошибки `io.EOF`.

Для каждой прочитанной строки вызывается метод `regexp.Regexp.ReplaceAllStringFunc()`, которому передаются строка текста и функция, реализующая замену. Затем вызовом метода `bufio.Writer.WriteString()` предпринимается попытка записать строку (возможно, модифицированную) – этот метод принимает строку для записи, записывает ее в виде последовательности байтов в кодировке UTF-8 и возвращает количество записанных байтов и значение ошибки (которое может быть равно `nil` в случае ее отсутствия). Нас не интересует количество записанных байтов, поэтому оно присваивается пустому идентификатору `_`. Если значение переменной `err` не равно `nil`, функция немедленно завершает работу, возвращая значение ошибки вызывающей программе.

Использование значений для чтения и записи, созданных с помощью пакета `bufio`, как в данном примере, позволяет работать с высокоуровневыми значениями типа `string` и полностью избавляет от сложностей обработки двоичных байтов, представляющих текст на диске. И, разумеется, благодаря отложенной *анонимной функции* можно быть уверенными, что все данные в буфере будут вытолкнуты из буфера в момент завершения функции `americanise()`, если в процессе ее работы не произошло никаких ошибок.

```
func makeReplacerFunction(file string) (func(string) string, error) {
    rawBytes, err := ioutil.ReadFile(file)
    if err != nil {
        return nil, err
    }
    text := string(rawBytes)
    usForBritish := make(map[string]string)
    lines := strings.Split(text, "\n")
    for _, line := range lines {
        fields := strings.Fields(line)
        if len(fields) == 2 {
            usForBritish[fields[0]] = fields[1]
        }
    }
    return func(word string) string {
        if usWord, found := usForBritish[word]; found {
            return usWord
        }
        return word
    }, nil
}
```

Функция `makeReplacerFunction()` принимает имя файла с оригинальными строками и строками замены и возвращает функцию, которая для оригинальных строк будет возвращать соответствующие им строки замены, а также значение ошибки `error`. Она предполагает, что файл содержит текст в кодировке UTF-8, где в каждой строке находятся оригинальное слово и слово замены, разделенные пробельными символами.

В дополнение к пакету `bufio` с его типами значений для чтения и записи в языке Go имеется также пакет `io/ioutil`, предоставляющий ряд высокоуровневых функций, включая используемую здесь функцию `ioutil.ReadFile()`. Она читает файл целиком и возвращает все его содержимое в виде последовательности двоичных байтов (`[]byte`) вместе со значением ошибки. Как обычно, если значение ошибки не равно `nil`, функция немедленно возвращает управление вызывающей программе со значением `nil` в качестве ссылки на функцию, реализующую замену. В случае успеха прочитанные байты преобразуются в строку с помощью инструкции преобразования вида `тип(переменная)`. Операция преобразования байтов UTF-8 в строку практически не имеет накладных расходов, потому что для внутреннего представления строк в языке Go используется кодировка UTF-8. (Подробнее о преобразовании строк рассказывается в главе 3.)

Функция, реализующая замену, которую требуется создать, должна принимать строку и возвращать соответствующую ей строку. Поэтому в функции необходимо реализовать своего рода поиск по таблице. Для этих целей идеально подходит тип данных `map` (§4.3). Тип `map` представляет собой коллекцию пар *ключ/значение* и обеспечивает очень быстрый поиск по *ключу*. Поэтому в данной функции британские слова будут играть роль ключей коллекции, а их американские аналоги – роль значений.

Отображения, срезы и каналы в языке Go создаются с помощью встроенной функции `make()`. Она создает значение указанного типа и возвращает ссылку на него. Эта ссылка может передаваться, например, другим функциям, и все изменения, произведенные в значении по этой ссылке, будут видимы в любом месте программы. Здесь создается пустое отображение с именем `usForBritish` со строковыми ключами и значениями.

После создания отображения содержимое текстового файла (представленное в форме одной длинной строки) разбивается на отдельные строки с помощью функции `strings.Split()`. Эта функция принимает строку для разбиения со строкой-разделителем, по

которой выполняется разбиение, и разбивает исходную строку на столько частей, сколько получится. (Если бы потребовалось ограничить количество разбиений, можно было бы использовать функцию `strings.SplitN()`.)

Для итераций по полученным строкам используется уже знакомый нам цикл `for`, но на этот раз – его разновидность, использующая предложение `range`. Данную форму удобно использовать для итераций по ключам и значениям отображений, по элементам каналов обмена данными или, как в данном случае, по элементам срезов (или массивов). Когда выполняется цикл по срезу (или массиву), в каждой итерации возвращается индекс (начиная с 0) и элемент с этим индексом (если срез непустой). В данном примере цикл `for` используется для обхода всех строк, но, поскольку значение индекса нас не интересует, он присваивается пустому идентификатору (`_`), то есть просто отбрасывается.

Каждую из строк требуется разбить на две строки: оригинальную строку и строку замены. Для этого можно было бы воспользоваться функцией `strings.Split()`, но тогда пришлось бы точно определить строку-разделитель, такую как " ", что могло бы приводить к ошибкам при работе с файлами, созданными вручную, где пользователь непреднамеренно может вставить несколько пробелов или символ табуляции. К счастью, в языке Go имеется функция `strings.Fields()`, разбивающая заданную строку по пробельным символам и потому более подходящая для работы с текстом, созданным человеком.

Если переменная `fields` (типа `[]string`) получает точно два элемента, они вставляются в виде пары *ключ/значение* в отображение. После заполнения отображения можно приступить к созданию функции, реализующей замену, которую необходимо вернуть вызывающей программе.

Функция, реализующая замену, создается в виде анонимной функции – аргумента инструкции `return`, которой также передается значение `nil` ошибки. (Разумеется, можно было бы написать менее компактный программный код, присвоив анонимную функцию переменной и возвращая эту переменную.) Функция имеет сигнатуру, в точности соответствующую требованиям метода `regexp.Regexp.ReplaceAllStringFunc()`, которому она передается.

Внутри анонимной функции, реализующей замену, просто выполняется поиск заданного слова. Если в отображении обнаруживается искомый ключ, соответствующее ему значение присваивается

первой переменной слева от оператора присваивания, в противном случае, если искомый ключ отсутствует в отображении, присваивается пустое значение. Но если пустое значение, возвращаемое оператором индексирования отображения, является допустимым значением, как узнать, что данный ключ отсутствует в отображении? В языке Go для этого можно использовать прием, удобный для случаев, когда необходимо определить наличие ключа в отображении, который заключается в том, чтобы поместить две переменные слева от оператора присваивания. Первой из них будет присваиваться значение, а второй – логическое значение, признак наличия искомого ключа. Именно этот прием и был использован в данном примере, где в инструкции `if` были совмещены инструкция сокращенного объявления переменных и проверка условия (значения логической переменной `found`). Таким образом, мы получаем значение `usWord` (которое будет пустой строкой в случае отсутствия искомого ключа в отображении) и флаг `found` типа `bool`. Если искомое британское слово было найдено в отображении, возвращается его американский эквивалент, иначе возвращается оригинальное слово.

В функции `makeReplacerFunction()` имеется одна тонкость, незаметная на первый взгляд. В созданной внутри нее анонимной функции выполняется обращение к отображению `usForBritish`, которое создается за пределами этой анонимной функции. Такое возможно благодаря поддержке *замыканий* в языке Go (§5.6.3). Замыкание – это функция, которая как бы «захватывает» окружение вызова, например состояние функции, внутри которой было создано замыкание, или, по крайней мере, часть окружения, используемая в функции-замыкании. То есть анонимная функция, созданная внутри функции `makeReplacerFunction()`, является замыканием, сохраняющим отображение `usForBritish`.

Другая тонкость заключается в том, что отображение `usForBritish` является локальной переменной и недоступно за пределами функции, где оно объявлено. *Локальные переменные* в языке Go с успехом могут играть роль возвращаемых значений. Даже если эти переменные являются ссылками или указателями, они не будут удаляться, пока используются, и немедленно будут утилизированы сборщиком мусора, как только выйдут из употребления (то есть когда все переменные, хранящие, ссылающиеся или указывающие на значение, выйдут из текущей области видимости).

В этом разделе были представлены некоторые основные низкоуровневые и высокоуровневые функции для работы с файлами, такие



как `os.Open()`, `os.Create()` и `ioutil.ReadFile()`. Тема работы с файлами подробно будет рассматриваться в главе 8, включая приемы чтения записи текста, двоичных данных, а также данных в формате JSON и XML. Встроенные типы коллекций, такие как срезы и отображения, в значительной степени избавляют от необходимости определять собственные типы данных и обеспечивают чрезвычайно высокую производительность и удобство в использовании. Типы коллекций в языке Go рассматриваются в главе 4. Интерпретация функций как обычных значений и поддержка замыканий делают возможным использование некоторых передовых и весьма мощных идиом программирования. А инструкция `defer` обеспечивает простой механизм устранения утечек ресурсов.

1.7. Из полярных координат в декартовы – параллельное программирование

Одним из ключевых аспектов языка Go является возможность использовать преимущества современных компьютеров на аппаратной архитектуре с несколькими процессорами или ядрами, без обременения программистов большим количеством шаблонных операций. Многие многопоточные программы могут быть написаны на языке Go вообще без явного использования механизма блокировок (хотя в действительности в Go имеются блокировки, которые при необходимости можно использовать в низкоуровневом программном коде, как будет показано в главе 7).

В языке Go имеются две особенности, превращающие параллельное программирование в удовольствие. Первая: *go-подпрограммы* (*goroutines*), фактически очень легковесные потоки выполнения/сопрограммы, могут быть легко созданы, без необходимости определять подклассы некоторого класса, инкапсулирующего функциональность потоков выполнения (что, впрочем, в языке Go в принципе невозможно). Вторая: каналы, обеспечивающие надежное средство одно- и двустороннего обмена данными между *go-подпрограммами* и которые можно использовать для их синхронизации.

Многопоточная модель выполнения в языке Go основана на *обмене данными*, а не на их совместном использовании. Это значительно упрощает создание многопоточных программ, по сравнению с традиционными подходами, основанными на применении потоков

выполнения и механизма блокировок, поскольку при отсутствии совместно используемых данных невозможно попасть в состояние гонки за ресурсами (или взаимоблокировки) и нет необходимости приобретать или освобождать блокировки, так как нет нужды защищать доступ к совместно используемым данным.

В этом разделе будет рассмотрен пятый и последний пример главы «обзора» примеров. Здесь будет представлен пример программы, использующей два канала обмена данными и реализующей их обработку в отдельной го-подпрограмме. Для такой маленькой программы это слишком сложный подход, но ее главная цель – продемонстрировать основы использования этих возможностей языка Го максимально простым и коротким способом. Более практичные примеры параллельного программирования, демонстрирующие множество разных приемов работы с каналами и го-подпрограммами, будут представлены в главе 7.

Программа, которая будет рассматриваться здесь, называется `polar2cartesian`. Это диалоговая консольная программа, предлагающая пользователю ввести два числа, разделенных пробелами, радиус и угол, и вычисляющая соответствующие им декартовы координаты точки. Кроме того, для иллюстрации одного из подходов к реализации параллельной обработки данных здесь также демонстрируются некоторые простые структуры и порядок определения типа операционной системы, Unix-подобной или Windows, в которой выполняется программа, когда эти различия имеют значение. Ниже приводится пример сеанса работы с программой в консоли Linux:

```
$ ./polar2cartesian
```

```
Enter a radius and an angle (in degrees), e.g., 12.5 90, or Ctrl+D to quit.
```

```
Radius and angle: 5 30.5
```

```
Polar radius=5.00 q=30.50° @ Cartesian x=4.31 y=2.54
```

```
Radius and angle: 5 -30.25
```

```
Polar radius=5.00 q=-30.25° @ Cartesian x=4.32 y=-2.52
```

```
Radius and angle: 1.0 90
```

```
Polar radius=1.00 q=90.00° @ Cartesian x=-0.00 y=1.00
```

```
Radius and angle: ^D
```

```
$
```

Исходный текст программы находится в файле `polar2cartesian/polar2cartesian.go`. Мы рассмотрим его содержимое сверху вниз, начав с инструкции импортирования, затем перейдем к объявлению структур, функции `init()`, функции `main()`, потом – к функциям, которые вызываются функцией `main()`, и т. д.

```
import (  
    "bufio"  
    "fmt"  
    "math"  
    "os"  
    "runtime"  
)
```

Программа `polar2cartesian` импортирует несколько пакетов, часть которых уже знакома нам по предыдущим разделам, поэтому упомянуты будут только новые пакеты. Пакет `math` предоставляет математические функции для работы с вещественными числами (§2.3.2), а пакет `runtime` – функции для доступа к свойствам программы времени выполнения, таким как тип платформы, на которой выполняется программа.

```
type polar struct {  
    radius float64  
    θ      float64  
}  
type cartesian struct {  
    x float64  
    y float64  
}
```

Ключевое слово `struct` в языке Go обозначает тип, хранящий (агрегирующий, или встраивающий) одно или более полей. Поля могут быть любого типа, встроенного, как здесь (`float64`), структурами, интерфейсами или их комбинациями. (Поля, имеющие тип интерфейса, фактически являются указателями на значения любого типа, поддерживающие указанный интерфейс, то есть реализующие методы, определяемые интерфейсом.)

Для обозначения угла в полярных координатах естественным кажется использовать символ тета (θ) греческого алфавита, и благодаря поддержке кодировки UTF-8 в языке Go это вполне возможно. То есть язык Go позволяет использовать в идентификаторах любые буквы Юникода, а не только символы латиницы.

Несмотря на то, что две структуры содержат поля одних и тех же типов, они являются разными типами данных, и значения этих типов не могут автоматически преобразовываться друг в друга. Благодаря этому обеспечивается надежность. В конечном счете простая замена

полярных координат декартовыми не имеет смысла. В некоторых ситуациях подобные преобразования могут быть вполне оправданы, особенно когда легко можно создать метод преобразования (то есть метод, принимающий значение одного типа и возвращающий значение другого типа), использующий синтаксис *составных литералов* для создания значения целевого типа, заполненного значениями полей из исходного типа. (Преобразование данных числовых типов рассматривается в главе 2, а преобразование данных строковых типов – в главе 3.)

```
var prompt = "Enter a radius and an angle (in degrees), e.g., 12.5 90, " +
    "or %s to quit."
func init() {
    if runtime.GOOS == "windows" {
        prompt = fmt.Sprintf(prompt, "Ctrl+Z, Enter")
    } else { // Unix-подобная система
        prompt = fmt.Sprintf(prompt, "Ctrl+D")
    }
}
```

Если в пакете имеется одна или более функций `init()`, они автоматически будут вызваны до вызова функции `main()` в пакете `main`. (В действительности функции `init()` не требуется вызывать явно.) Поэтому при запуске программы `polar2cartesian` первой будет вызвана эта функция `init()`. Функция `init()` определяет содержимое строки приглашения к вводу, учитывая различия в обозначении конца файла на разных платформах. Например, в Windows признак конца файла можно ввести, нажав комбинацию клавиш **Ctrl+Z** и затем **Enter**. В пакете `runtime` имеется константа `GOOS` (Go Operating System), являющаяся строкой, идентифицирующей операционную систему, в которой выполняется программа. Типичными значениями константы являются строки `darwin` (Mac OS X), `freebsd`, `linux` и `windows`.

Прежде чем перейти к исследованию функции `main()` и остальной части программы, обсудим коротко каналы обмена данными и рассмотрим несколько игрушечных примеров их использования.

Каналы в языке Go имитируют каналы операционной системы Unix и обеспечивают двусторонний (или, при желании, односторонний) обмен данными. Каналы действуют подобно очередям FIFO (First In, First Out – первый пришел, первый ушел), то есть они сохраняют порядок следования элементов данных, передаваемых через них. Элементы нельзя просто так выбросить из канала, но их можно игнорировать при приеме. Рассмотрим очень простой пример. Сначала создадим канал:



```
messages := make(chan string, 10)
```

Каналы создаются с помощью функции `make()` (глава 7) и объявляются с помощью синтаксической конструкции `chan` Тип. Инструкция выше создаст канал, позволяющий передавать и принимать строки. Вторым аргументом функции `make()` определяется размер буфера (по умолчанию равен 0) – в данном случае выделяется буфер, способный хранить до десяти строк. Если буфер канала оказывается заполненным до конца, доступ к нему блокируется, пока из канала не будет извлечен хотя бы один элемент. Это означает, что через канал может передаваться любое количество элементов, при условии что они своевременно будут извлекаться из канала, чтобы освободить место для последующих элементов. Канал, в котором буфер имеет нулевой размер, может использоваться для отправки значений, только когда на другом конце канала ожидается прием данных. (Эффекта неблокирующих каналов можно добиться с помощью инструкции `select`, как будет показано в главе 7.)

Теперь отправим пару строк в канал:

```
messages <- "Leader"  
messages <- "Follower"
```

Когда оператор передачи данных `<-` используется в роли двухместного оператора, левым операндом должен быть канал, а правым – значение, отправляемое в канал, типа указанного при объявлении канала. Здесь первой в канал отправляется строка `Leader`, а затем – строка `Follower`.

```
message1 := <-messages  
message2 := <-messages
```

Когда оператор передачи данных `<-` используется в роли унарного оператора, с единственным правым операндом (который должен быть каналом), он осуществляет прием данных из канала, блокируя выполнение программы, пока из канала не будет извлечено значение. В этом примере из канала `messages` извлекаются два сообщения. Переменной `message1` будет присвоена строка `Leader`, а переменной `message2` – строка `Follower`; обе переменные имеют тип `string`.

Обычно каналы создаются с целью обеспечить взаимодействия между `go`-подпрограммами. При выполнении операций отправки и получения данных не требуется использовать механизм блокировок,

а блокирующее поведение каналов можно использовать для *синхронизации*.

Теперь, познакомившись с некоторыми основными особенностями каналов, рассмотрим практическое их использование в го-подпрограммах.

```
func main() {
    questions := make(chan polar)
    defer close(questions)
    answers := createSolver(questions)
    defer close(answers)
    interact(questions, answers)
}
```

Как только выполнятся все функции `init()`, будет вызвана функция `main()` из пакета `main`.

Здесь функция `main()` сначала создает канал (типа `chan polar`) для передачи значений типа `polar struct` и сохраняет его в переменной `questions`. После создания канала сразу же выполняется инструкция `defer`, вызывающая встроенную функцию `close()` (см. табл. 5.1 ниже), чтобы обеспечить его закрытие, когда он станет ненужным. Далее вызывается функция `createSolver()`. Она принимает канал `questions` и возвращает соответствующий ему канал `answers` (типа `chan cartesian`). Далее следует вторая инструкция `defer`, обеспечивающая закрытие канала `answers` при выходе из функции. И наконец, вызывается функция `interact()`, которой передаются оба канала и которая реализует взаимодействие с пользователем.

```
func createSolver(questions chan polar) chan cartesian {
    answers := make(chan cartesian)
    go func() {
        for {
            polarCoord := <-questions ❶
            θ := polarCoord.θ * math.Pi / 180.0 // преобразование градусов в
радианы
            x := polarCoord.radius * math.Cos(θ)
            y := polarCoord.radius * math.Sin(θ)
            answers <- cartesian{x, y} ❷
        }
    }()
    return answers
}
```

Функция `createSolver()` сначала создает канал `answers`, посредством которого будут отправляться ответы (то есть декартовы координаты) на вопросы (то есть полярные координаты), принимаемые из канала `questions`.

За созданием канала следует инструкция `go`. Этой инструкции передается вызов функции (синтаксически инструкция `go` подобна инструкции `defer`), которая будет выполняться в отдельной, асинхронной `go`-подпрограмме. Это означает, что выполнение текущей функции (то есть главной `go`-подпрограммы) будет немедленно продолжено с инструкции, следующей за объявлением функции. В данном случае за инструкцией `go` следует инструкция `return`, возвращающая канал `answers` вызывающей программе. Как отмечалось выше, в языке Go допускается возвращать локальные переменные, поскольку все бремя управления памятью берет на себя язык Go.

В данном случае в инструкции `go` создается (и вызывается) анонимная функция. Функция выполняет бесконечный цикл, ожидающий (то есть блокирующий выполнение собственной `go`-подпрограммы, но не других `go`-подпрограмм и не функции, в которой `go`-подпрограмма была запущена) появления запросов в канале `questions`, в данном случае значений типа `polar struct`. Когда в канале появятся полярные координаты, анонимная функция вычислит соответствующие им декартовы координаты (с помощью пакета `math` из стандартной библиотеки) и отправит ответ типа `cartesian struct` (созданный с помощью синтаксиса *составных литералов*) в канал `answers`.

В инструкции ❶ используется унарный оператор `<-`, извлекающий полярные координаты из канала `questions`. А в инструкции ❷ используется двухместная версия оператора `<-`, левым операндом которого является канал `answers`, куда отправляются данные, а правым операндом – отправляемое значение `cartesian`.

Как только функция `createSolver()` вернет управление, два настроенных канала обмена данными и отдельная `go`-подпрограмма, ожидающая получения полярных координат из канала `questions`, при этом выполнение всех остальных `go`-подпрограмм, включая главную, в которой выполняется функция `main()`, не блокируется.

```
const result = "Polar radius=%.02f q=%.02f° @ Cartesian x=%.02f y=%.02f\n"
func interact(questions chan polar, answers chan cartesian) {
    reader := bufio.NewReader(os.Stdin)
```



```
fmt.Println(prompt)
for {
    fmt.Printf("Radius and angle: ")
    line, err := reader.ReadString('\n')
    if err != nil {
        break
    }
    var radius, θ float64
    if _, err := fmt.Sscanf(line, "%f %f", &radius, &θ); err != nil {
        fmt.Fprintln(os.Stderr, "invalid input")
        continue
    }
    questions <- polar{radius, θ}
    coord := <-answers
    fmt.Printf(result, radius, θ, coord.x, coord.y)
}
fmt.Println()
}
```

При вызове этой функции в виде параметров ей передаются оба канала. Она начинается с создания буферизованного значения чтения для `os.Stdin`, необходимого для взаимодействия с пользователем через консоль. Затем она выводит приглашение к вводу, сообщаящее, как выполнить ввод и как завершить программу. Вместо того чтобы предлагать пользователю ввести признак конца файла, можно было бы предусмотреть завершение программы по простому нажатию клавиши **Enter** (то есть при отсутствии каких-либо значений). Однако требование ввести признак конца файла делает программу `polar2cartesian` более гибкой, так как это дает возможность организовать чтение данных из произвольного внешнего файла с применением операции перенаправления (предполагается, что в каждой строке этот файл содержит два числа, разделенных пробельными символами).

Затем функция начинает выполнять бесконечный цикл, который в каждой итерации предлагает пользователю ввести полярные координаты (радиус и угол). После этого функция ждет, пока пользователь завершит ввод и нажмет клавишу **Enter** или **Ctrl+D** (**Ctrl+Z** в Windows). Здесь не предусматривается обработка ошибок – если в процессе ввода возникла какая-либо ошибка, функция просто возвращает управление вызывающей программе (функции `main()`), которая, в свою очередь, вернет управление (и выполнит отложенные вызовы в инструкциях `defer`, чтобы закрыть каналы).

В случае успеха создаются две переменные типа `float64` для хранения чисел, введенных пользователем, и затем вызывается функция `fmt.Sscanf()`, выполняющая разбор строки. Эта функция принимает исходную строку, формат – в данном случае два вещественных числа, разделенных пробелом, – и один или более указателей на переменные, куда должны быть записаны результаты разбора строки. (Оператор `&` получения адреса применяется для создания указателя на значение, подробнее о нем рассказывается в §4.1.) Функция возвращает количество элементов, которые ей успешно удалось вычленивать из исходной строки, и значение ошибки (или `nil`). В случае ошибки функция выводит соответствующее сообщение в устройство `os.Stderr`, что обеспечивает вывод сообщения в консоли, даже если устройство `os.Stdout` программы будет перенаправлено в файл. Подробнее о мощных и гибких функциях сканирования в языке Go рассказывается в главе 8 (§8.1.3.2), а полный их перечень приводится в табл. 8.2.

Если пользователь ввел допустимые числовые значения и они успешно были отправлены в канал `questions` (в виде значения типа `polar struct`), выполнение главной `go`-подпрограммы блокируется до появления ответа в канале `answers`. Дополнительная `go`-подпрограмма, созданная в функции `createSolver()`, находится в заблокированном состоянии, ожидая появления полярных координат в канале `questions`, поэтому, когда полярные координаты будут отправлены, эта дополнительная `go`-подпрограмма выполнит вычисления, отправит получившиеся декартовы координаты в канал `answers` и остановится в ожидании следующих координат. Как только декартовы координаты окажутся в канале `answers`, выполнение функции `interact()` будет продолжено. Она выведет полученные результаты с помощью функции `fmt.Printf()`, передав ей исходные полярные и полученные декартовы координаты в виде аргументов, которые будут подставлены вместо спецификаторов формата `%`. Порядок взаимодействий между `go`-подпрограммами посредством каналов иллюстрирует рис. 1.1.

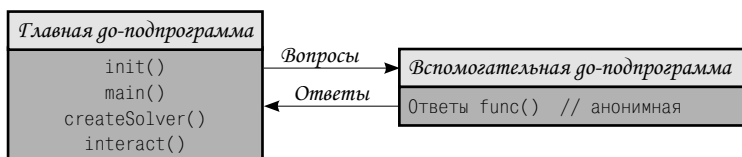


Рис. 1.1. Две взаимодействующие `go`-подпрограммы

Цикл `for` в функции `interact()` является бесконечным, поэтому сразу после вывода результатов пользователю вновь будет предложено ввести радиус и угол. Выполнение цикла прекращается, только когда будет прочитан признак конца файла – либо введенный пользователем, либо по достижении конца входного файла, при использовании операции перенаправления.

Вычисления в программе `polar2cartesian` не отличаются сложностью, поэтому здесь нет действительной необходимости производить их в отдельной `go`-подпрограмме. Однако в аналогичных программах, производящих массу тяжеловесных расчетов после каждого ввода, можно было бы извлечь немалую выгоду от подобной организации вычислений, например задействовав по одной `go`-подпрограмме на каждую группу введенных значений. Более практичные примеры использования каналов и `go`-подпрограмм будут представлены в главе 7.

На этом заканчивается обзор возможностей языка `Go` на пяти примерах программ, представленных в этой главе. Естественно, возможности языка `Go` гораздо шире, чем было показано здесь и в чем вы убедитесь в следующих главах, каждая из которых посвящена определенному аспекту языка и соответствующим пакетам из стандартной библиотеки. Эта глава завершается небольшим упражнением, которое, несмотря на свой размер, требует вдумчивости и внимательности.

1.8. Упражнение

Скопируйте содержимое каталога `bigdigits` в каталог, например, `my_bigdigits` и измените `my_bigdigits/bigdigits.go` так, чтобы новая версия программы `bigdigits` (см. §1.4) могла по желанию пользователя выводить строки из символов «*» над и под изображением числа, добавьте обработку соответствующего аргумента командной строки.

Оригинальная программа выводит сообщение с инструкцией по использованию, если не было указано число для вывода. Добавьте вывод этого сообщения, если пользователь указал аргумент `-h` или `--help`. Например:

```
$ ./bigdigits --help
usage: bigdigits [-b|--bar] <whole-number>
-b --bar draw an underbar and an overbar
```

Если аргумент `--bar` (или `-b`) не указан пользователем, программа должна действовать, как и прежде. Ниже приводится пример вывода программы при наличии аргумента `--bar` (или `-b`):

```
$ ./bigdigits --bar 8467243
```

```
*****
888      4      666  77777  222      4      333
8  8      44      6          7  2  2      44      3  3
8  8      4  4      6          7      2      4  4      3
888      4  4      6666      7          2      4  4      33
8  8      444444      6  6      7          2      444444      3
8  8          4      6  6  7          2          4      3  3
888          4      666  7          22222      4      333
*****
```

Для решения этого упражнения потребуется реализовать более сложную обработку аргументов командной строки, чем в оригинальной версии, и добавить незначительные изменения в программный код, производящий вывод, предусмотрев печать строки из символов «*» перед первой и после последней строки изображения числа. В целом для решения упражнения потребуется добавить около 20 строк программного кода – размер функции `main()` в новой версии программы увеличится вдвое, по сравнению с оригиналом (~40 против ~20 строк), причем основной прирост будет приходиться на реализацию обработки аргументов командной строки. Пример решения этого упражнения приводится в файле `bigdigits_ans/bigdigits.go`.

Подсказка: это решение также немного иначе конструирует строки с изображением числа, чтобы строки из звездочек не получились длиннее самого изображения. Кроме того, новая версия импортирует пакет `strings` и использует функцию `strings.Repeat(string, int)`. Эта функция возвращает строку, сконструированную из строки `string` в первом аргументе, повторяющейся число раз, определяемое вторым аргументом `int`. Ознакомьтесь с описанием этой функции в локальной документации (см. врезку «Документация по языку Go» в начале главы) или на странице golang.org/pkg/strings и начните знакомиться с документацией к стандартной библиотеке языка Go.

Реализовать обработку аргументов командной строки можно намного проще, если задействовать пакет, специально предназначенный для этого. В состав стандартной библиотеки Go входит достаточно простой пакет разбора аргументов командной строки, `flag`, поддерживающий обработку параметров в стиле X11 (таких как `-option`). Кроме того, существует несколько пакетов, поддерживающих разбор коротких и длинных аргументов командной строки в стиле GNU (таких как `-o` и `--option`), доступных на странице go-dashboard.appspot.com/project.



2. Логические значения и числа

Это первая из четырех глав, посвященных процедурному программированию и закладывающих основы программирования на языке Go – процедурного, объектно-ориентированного, параллельного и различных их комбинаций.

Эта глава охватывает встроенный логический тип и все встроенные числовые типы, а также рассматривает два числовых типа из стандартной библиотеки Go. Помимо встроенного типа комплексных чисел и необходимости явно выполнять преобразования из одного числового типа в другой, программисты, пришедшие из мира C, C++ и Java, найдут в этой главе несколько неожиданных сюрпризов.

В первом разделе этой главы рассматриваются некоторые основы языка, такие как порядок создания комментариев, ключевые слова и операторы, допустимые идентификаторы и т. д. Вслед за разделом с начальными сведениями следуют параграфы, описывающие логические значения, целые и вещественные числа и, наконец, комплексные числа.

2.1. Начальные сведения

В языке Go поддерживаются два типа *комментариев*, оба заимствованные из языка C++. Однострочные комментарии начинаются с символов `//` и заканчиваются символом перевода строки. Они интерпретируются просто как символ перевода строки. Универсальные комментарии начинаются символами `/*` и заканчиваются символами `*/`, могут располагаться на нескольких строках. Когда универсальный комментарий целиком располагается внутри строки (например, `/* комментарий на одной строке */`), он интерпретируется как пробел, а когда целиком занимает одну или более строк – как *символ перевода строки*. (Символы перевода строки имеют большое значение в языке Go, как будет показано в главе 5.)

Идентификаторами в языке Go могут быть непустые последовательности букв и цифр, начинающиеся с буквы и не совпадающие

с ключевыми словами. Буквами считаются символ подчеркивания `_` и все символы из категорий Юникода: «Lu» (буквы верхнего регистра), «Ll» (буквы нижнего регистра), «Lt» (заглавные буквы), «Lm» (буквы-модификаторы) или «Lo» (прочие буквы); включая все алфавитные символы английского алфавита (A–Z и a–z). Цифрами считаются любые символы из категории Юникода «Nd» (числа, десятичные цифры); включая арабские цифры (0–9). Компилятор не позволяет использовать идентификаторы, совпадающие с ключевыми словами, перечисленными в табл. 2.1.

Таблица 2.1. Ключевые слова в языке Go

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

В языке Go имеется множество predefined идентификаторов. В программе допускается создавать собственные идентификаторы с именами, совпадающими с именами predefined идентификаторов, перечисленных в табл. 2.2, хотя это и нецелесообразно.

Таблица 2.2. Predefined идентификаторы в языке Go

append	copy	int8	nil	true
bool	delete	int16	panic	uint
byte	error	int32	print	uint8
cap	false	int64	println	uint16
close	float32	iota	real	uint32
complex	float64	len	recover	uint64
complex64	imag	make	rune	uintptr
complex128	int	new	string	

Идентификаторы чувствительны к регистру символов, то есть идентификаторы `LINECOUNT`, `linecount`, `LineCount`, `lineCount` и `linecount`, считаются разными идентификаторами. Идентификаторы, начинающиеся с буквы в верхнем регистре, то есть с буквы из категории Юникода «Lu» (включая символы A–Z), считаются общедоступными, или *экспортируемыми*, в терминологии Go, а все

остальные – частными, или *неэкспортируемыми*. (Это правило не применяется к именам пакетов, которые в соответствии с соглашениями состоят только из букв в нижнем регистре.) Подробнее об этих различиях будет рассказываться при обсуждении особенностей объектно-ориентированного программирования в главе 6 и исследовании пакетов в главе 9.

Пустой идентификатор _ играет роль временного идентификатора в операциях присваивания, где ожидается переменная, и собирает любые присваиваемые ему значения. Пустой идентификатор не может считаться новой переменной, поэтому при использовании вместе с оператором := слева должна быть указана хотя бы одна другая (новая) переменная. Допускается отбрасывать некоторые или все значения, возвращаемые функцией, присваивая их пустому идентификатору. Однако, если программе не нужно ни одно из возвращаемых значений, гораздо проще просто игнорировать их. Например:

```
count, err = fmt.Println(x) // получить число напечатанных байтов и ошибку
count, _ = fmt.Println(x)   // получить число напечатанных байтов; отбросить ошибку
_, err = fmt.Println(x)     // отбросить число напечатанных байтов; получить ошибку
fmt.Println(x)              // игнорировать все возвращаемые значения
```

При выводе данных в консоль общепринято игнорировать возвращаемые значения, но при выводе в файлы, сетевые соединения и т. д., с помощью функции `fmt.Fprint()` и подобных ей, всегда следует проверять значение ошибки. (Детальное описание функций вывода в языке Go приводится в §3.5.)

2.1.1. Константы и переменные

Константы объявляются с помощью ключевого слова `const`. Переменные могут объявляться с помощью ключевого слова `var` или оператора сокращенного объявления переменных. Компилятор Go способен определять тип объявляемой переменной по выражению справа от оператора присваивания. Однако при желании или необходимости допускается явно указывать тип, например если требуется указать иной тип, отличающийся от типа, определяемого компилятором автоматически. Ниже приводятся несколько примеров объявлений:

```
const limit = 512           // константа; совместима с любыми числовыми типами
const top uint16 = 1421     // константа; тип: uint16
start := -19                // переменная; определяемый компилятором тип: int
```

```
end := int64(9876543210) // переменная; тип: int64
var i int                // переменная; значение 0; тип: int
var debug = false        // переменная; определяемый компилятором тип: bool
checkResults := true     // переменная; определяемый компилятором тип: bool
stepSize := 1.5          // переменная; определяемый компилятором тип: float64
acronym := "FOSS"        // переменная; определяемый компилятором тип: string
```

Для целочисленных литералов компилятор Go определяет тип `int`, для вещественных литералов – тип `float64`, а для литералов комплексных чисел – тип `complex128` (число в названиях типов отражает количество бит, занимаемых значением этого типа). Обычной практикой считается не указывать тип явно, если только не требуется использовать какой-то конкретный тип, который не определяется компилятором, – подробнее об этом рассказывается в §2.3. Типизированные числовые константы (такие как `top` в примере выше) могут использоваться только в выражениях с числовыми значениями того же типа (если явно не преобразовывать их тип). Нетипизированные числовые константы могут использоваться в выражениях с числовыми значениями любых встроенных типов (например, константу `limit` можно использовать в выражениях с целыми или вещественными числами).

Переменной `i` в примере выше не было явно присвоено какое-либо значение. Это не влечет за собой никаких неожиданностей, потому что в языке Go переменным всегда присваиваются *нулевые значения*, соответствующие их типам, если явно не было присвоено другое значение. То есть числовая переменная гарантированно будет инициализирована нулем, а строковая – пустой строкой, если явно не указано другое значение. Благодаря этому устраняется проблема появления бессмысленных значений в неинициализированных переменных, от которой страдают некоторые другие языки.

2.1.1.1. Перечисления

Вместо того чтобы многократно повторять ключевое слово `const`, когда требуется определить несколько констант, их можно сгруппировать в одно объявление с единственным ключевым словом `const`. (Аналогичный синтаксис группировки уже демонстрировался в главе 1, где он использовался для объявления импортируемых пакетов; эту же синтаксическую конструкцию можно использовать для группировки объявлений переменных с ключевым словом `var`.) В случаях, когда требуется лишь объявить константы с отличающимися

значениями, и при этом сами значения не играют никакой роли, можно воспользоваться поддержкой *перечислений*.

const Cyan = 0		const (const (
const Magenta = 1		Cyan = 0		Cyan = iota // 0
const Yellow = 2		Magenta = 1		Magenta // 1
		Yellow = 2		Yellow // 2
))

Эти три фрагмента достигают одной и той же цели. В сгруппированном объявлении констант первой из них присваивается нулевое значение, если явно не было указано другое (определенное значение или *iota*), а второй и всем остальным — значение предшествующей константы или *iota*, если значением предшествующей константы является *iota*. При этом для каждой последующей константы значение *iota* оказывается на единицу больше, чем для предшествующей.

Строго говоря, *iota* — это предопределенный идентификатор, представляющий *последовательность* нетипизированных целочисленных значений. Значение этого идентификатора сбрасывается в нуль при встрече каждого нового ключевого слова *const* (то есть каждый раз, когда встречается новое сгруппированное объявление констант) и увеличивается для каждой последующей константы в группе. Поэтому в правом фрагменте выше *все* константы получают значение *iota* (причем константы *Magenta* и *Yellow* — неявно). А так как объявление константы *Cyan* следует сразу за ключевым словом *const*, значение *iota* будет сброшено в нуль и присвоено константе *Cyan*. Константа *Magenta* также получит значение *iota*, но на этот раз оно будет увеличено на 1. Аналогично константа *Yellow* получит значение *iota*, которое к этому моменту станет равно 2. Если в конец этого фрагмента (внутри группового объявления) добавить константу *Black*, она неявно получит значение *iota*, но уже равное 3.

С другой стороны, если в правом фрагменте *убрать* явное присваивание значения *iota*, константа *Cyan* получит значение 0, константа *Magenta* получит значение константы *Cyan*, и константа *Yellow* получит значение константы *Magenta* — то есть все они получают значение 0. Аналогично, если константе *Cyan* присвоить значение 9, все константы в этой группе получат значение 9. Или, если константе *Magenta* присвоить значение 5, константа *Cyan* получит значение 0 (так как она является первой в группе и ей явно не назначено какое-либо другое значение или значение *iota*), константа *Magenta* получит значение 5 (указанное явно), и константа *Yellow* также получит значение 5 (значение предыдущей константы).

Значение `iota` можно еще использовать в комбинации с вещественными числами, простыми выражениями и пользовательскими типами.

```
type BitFlag int
const (
    Active BitFlag = 1 << iota // 1 << 0 == 1
    Send      // Неявно BitFlag = 1 << iota // 1 << 1 == 2
    Receive   // Неявно BitFlag = 1 << iota // 1 << 2 == 4
)
flag := Active | Send
```

В этом фрагменте создаются три битовых флага пользовательского типа `BitFlag`, и затем переменной `flag` (типа `BitFlag`) присваивается результат поразрядной операции ИЛИ двух из них (то есть переменная `flag` получит значение 3; поразрядные операции, поддерживаемые в языке Go, перечислены в табл. 2.6 ниже). В данном случае можно было бы опустить пользовательский тип в объявлениях констант, и тогда были бы созданы нетипизированные целочисленные константы, а для переменной `flag` был бы определен тип `int`. Переменным типа `BitFlag` можно присваивать любые значения типа `int`, тем не менее `BitFlag` — это совершенно другой тип, и значения этого типа могут участвовать в выражениях с числами типа `int`, только если они явно будут приводиться к типу `int` (или значения типа `int` будут явно приводиться к типу `BitFlags`).

Тип `BitFlag` имеет определенную практическую ценность, но его использование может вызвать некоторые сложности во время отладки. Если сейчас попробовать вывести значение переменной `flag`, мы просто получим число 3, без дополнительных подсказок о том, что оно означает. В языке Go легко можно изменить вывод значений пользовательских типов, потому что функции вывода из пакета `fmt` используют метод `String()` типа, если он определен. То есть, чтобы сделать вывод значений типа `BitFlag` более информативным, достаточно просто добавить в него соответствующий метод `String()`. (Подробно пользовательские типы и методы рассматриваются в главе 6.)

```
func (flag BitFlag) String() string {
    var flags []string
    if flag&Active == Active {
        flags = append(flags, "Active")
    }
```

```

}
if flag&Send == Send {
    flags = append(flags, "Send")
}
if flag&Receive == Receive {
    flags = append(flags, "Receive")
}
if len(flags) > 0 {
    // приведение типа int(flag) совершенно необходимо,
    // чтобы избежать бесконечной рекурсии!
    return fmt.Sprintf("%d(%)", int(flag), strings.Join(flags, "|"))
}
return "0()"
}

```

Этот метод создает срез (возможно, пустой) со строками для установленных битовых полей и выводит значение переменной в виде десятичного целого числа и строк, описывающих это значение. (Значение легко можно вывести в двоичном виде, заменив спецификатор формата `%d` на `%b`.) Как отмечено в комментарии, преобразование типа значения `flag` (типа `BitFlag`) в тип `int` в вызове функции `fmt.Sprintf()` играет важную роль, в противном случае произойдет рекурсивный вызов метода `BitFlag.String()`, что приведет к бесконечной рекурсии. (Встроенная функция `append()` описывается в §4.2.3, а функции `fmt.Sprintf()` и `strings.Join()` — в главе 3.)

```

Println(BitFlag(0), Active, Send, flag, Receive, flag|Receive)
0() 1(Active) 2(Send) 3(Active|Send) 4(Receive) 7(Active|Send|Receive)

```

Данный фрагмент демонстрирует, как выглядит вывод значения типа `BitFlags` с методом `String()`, — очевидно, что при отладке такой вывод будет полезнее, чем простое целое число.

Разумеется, вполне возможно создать пользовательский тип, представляющий ограниченный диапазон целых чисел, определив более сложное перечисление. Подробнее о пользовательских типах рассказывается в главе 6. Минималистский подход к определению перечислений полностью соответствует философии языка Go: он старается удовлетворить все потребности программистов, включая множество мощных и удобных особенностей, сохраняя при этом простоту и непротиворечивость синтаксиса и максимально высокую скорость (сборки и выполнения программ).

2.2. Логические значения и выражения

В языке Go имеются два встроенных логических значения, `true` и `false`, оба относятся к типу `bool`. Кроме того, в Go поддерживаются стандартные логические операторы и операторы сравнения, возвращающие результат типа `bool`, – все они перечислены в табл. 2.3.

Как будет показано в главе 5, логические значения и выражения используются в инструкциях `if` (§5.2.1), в условных выражениях инструкций `for` (§5.3) и иногда в предложениях `case` инструкции `switch` (§5.2.2).

К выражениям с двухместными логическими операторами (`||` и `&&`) применяется сокращенный порядок вычислений. Например, если в выражении `b1 || b2` подвыражение `b1` вернет значение `true`, тогда результатом всего выражения будет `true` независимо от значения подвыражения `b2`, поэтому программе будет возвращено значение `true` и подвыражение `b2` вычисляться не будет. Аналогично, если в выражении `b1 && b2` подвыражение `b1` вернет `false`, результатом всего выражения будет `false`, поэтому программе будет возвращено значение `false` и подвыражение `b2` вычисляться не будет.

Операторы сравнения в языке Go (`<`, `<=`, `==`, `!=`, `>=`, `>`) накладывают определенные ограничения на сравниваемые значения. Два значения должны иметь один и тот же тип или, если они являются интерфейсами, должны содержать реализацию одного и того же интерфейса. Если одно из значений является константой, его тип должен быть совместим с типом другого значения. То есть нетипизированные числовые константы можно сравнивать с числовыми значениями *любого* типа, но числовые значения разных типов, ни одно из которых не является константой, сравнивать нельзя, если явно не преобразовать тип одного операнда в тип другого. (Преобразование числовых типов обсуждается в §2.3.)

Операторы `==` и `!=` могут применяться к операндам любых совместимых типов, включая массивы и структуры, элементы которых могут сравниваться между собой с помощью `==` и `!=`. Эти операторы не могут использоваться для сравнения срезов, однако при необходимости такое сравнение можно выполнить с помощью функции `reflect.DeepEqual()` из стандартной библиотеки. Операторы `==` и `!=` можно использовать для сравнения двух указателей или двух интерфейсов, или для сравнения указателя, интерфейса или ссылки (например, на канал, отображение или срез) со значением `nil`. Остальные операторы сравнения (`<`, `<=`, `>=`, `>`) могут применяться только к

числам и строкам. (Поскольку в языке Go, подобно языкам C и Java, не поддерживается перегрузка операторов, для сравнения значений пользовательских типов необходимо предусматривать собственные методы или функции сравнения, такие как `Less()` или `Equal()`, как будет показано в главе 6.)

Таблица 2.3. Логические операторы и операторы сравнения

Оператор	Описание/результат
<code>!b</code>	Оператор логического отрицания. Вернет <code>false</code> , если <code>b</code> имеет значение <code>true</code>
<code>a b</code>	Оператор «логическое ИЛИ» с сокращенным порядком вычисления. Вернет <code>true</code> , если одно из подвыражений, <code>a</code> или <code>b</code> , вернет <code>true</code>
<code>a && b</code>	Оператор «логическое И» с сокращенным порядком вычисления. Вернет <code>true</code> , если оба подвыражения, <code>a</code> и <code>b</code> , вернут <code>true</code>
<code>x < y</code>	Вернет <code>true</code> , если значение выражения <code>x</code> меньше значения выражения <code>y</code>
<code>x <= y</code>	Вернет <code>true</code> , если значение выражения <code>x</code> меньше или равно значению выражения <code>y</code>
<code>x == y</code>	Вернет <code>true</code> , если значение выражения <code>x</code> равно значению выражения <code>y</code>
<code>x != y</code>	Вернет <code>true</code> , если значение выражения <code>x</code> не равно значению выражения <code>y</code>
<code>x >= y</code>	Вернет <code>true</code> , если значение выражения <code>x</code> больше или равно значению выражения <code>y</code>
<code>x > y</code>	Вернет <code>true</code> , если значение выражения <code>x</code> больше значения выражения <code>y</code>

2.3. Числовые типы

В Go имеется широкий диапазон встроенных числовых типов. Кроме того, стандартная библиотека добавляет поддержку целых значений типа `big.Int` и рациональных значений типа `big.Rat`, которые имеют неограниченный размер (то есть их размеры ограничиваются только доступным объемом машинной памяти). Все числовые типы считаются отличными друг от друга, то есть к числовым значениям разных типов (например, к значениям типов `int32` и `int`) нельзя применять двухместные арифметические операторы или операторы сравнения (такие как `+` или `<`). Нетипизированные числовые константы совместимы со всеми (встроенными) числовыми типами,

поэтому они могут складываться или сравниваться с другими числовыми значениями независимо от их типов (встроенных).

Чтобы выполнить арифметическую операцию или сравнить два числовых значения разных типов, необходимо выполнить преобразование типов, обычно к большему типу, дабы избежать потери точности. Синтаксис преобразования типа имеет вид: тип(значение), и там, где такие преобразования допустимы (то есть допускается преобразование одного числового типа в другой), они всегда преуспевают, даже если в результате преобразования происходит потеря данных. Например:

```
const factor = 3 // Константа factor совместима со всеми числовыми типами
i := 20000       // i автоматически получит тип int
i *= factor
j := int16(20)   // j получит тип int16; то же, что и: var j int16 = 20
i += int(j)      // Типы должны совпадать, поэтому преобразование обязательно
k := uint8(0)    // То же, что и: var k uint8
k = uint8(i)     // Успех, но k получит значение i, усеченное до 8 бит
fmt.Println(i, j, k) // Выведет: 60020 20 116
```

Если необходимо обеспечить преобразование к меньшему типу без потери данных, всегда можно реализовать подходящую функцию. Например:

```
func Uint8FromInt(x int) (uint8, error) {
    if 0 <= x && x <= math.MaxUint8 {
        return uint8(x), nil
    }
    return 0, fmt.Errorf("%d is out of the uint8 range", x)
}
```

Эта функция принимает аргумент типа `int` и возвращает значение типа `uint8` и `nil`, если целое число находится в заданном диапазоне, или `0` и значение ошибки в противном случае. Константа `math.MaxUint8` определена в пакете `math`, где также имеются похожие константы для остальных встроенных числовых типов. (Разумеется, для беззнаковых типов нет констант, определяющих минимальное значение, поскольку для всех таких типов минимальным является значение `0`.) Функция `fmt.Errorf()` возвращает значение ошибки, основанное на строке формата и указанном значении (или значениях). (О форматировании строк рассказывается в §3.5.)

Числовые значения одного типа могут сравниваться операторами сравнения (табл. 2.3 выше). Аналогично к значениям одного типа могут

применяться арифметические операторы – в табл. 2.4 перечислены операторы, которые могут применяться к значениям любых числовых (встроенных) типов, и в табл. 2.6 – только к целочисленным значениям.

Выражения, определяющие значения констант, вычисляются на этапе компиляции – в них могут использоваться любые арифметические и логические операторы, а также операторы сравнения. Например:

```
const (
    efri    int64 = 10000000000          // тип: int64
    hlutföllum    = 16.0 / 9.0          // тип: float64
    mælikvarða    = complex(-2, 3.5) * hlutföllum // тип: complex128
    erGjaldgengur = 0.0 <= hlutföllum && hlutföllum < 2.0 // тип: bool
)
```

В примере были использованы идентификаторы на исландском языке как напоминание о том, что Go полностью поддерживает идентификаторы на национальных языках. (Тип `complex()` будет описан чуть ниже, в §2.3.2.1.)

Таблица 2.4. Арифметические операторы, применимые ко всем встроенным числовым типам

Оператор	Описание/результат
<code>+x</code>	<code>x</code>
<code>-x</code>	Изменение знака <code>x</code>
<code>x++</code>	Увеличивает <code>x</code> на значение нетипизированной константы 1
<code>x--</code>	Уменьшает <code>x</code> на значение нетипизированной константы 1
<code>x += y</code>	Увеличивает значение <code>x</code> на значение <code>y</code>
<code>x -= y</code>	Уменьшает значение <code>x</code> на значение <code>y</code>
<code>x *= y</code>	Присваивает переменной <code>x</code> результат умножения <code>x</code> на <code>y</code>
<code>x /= y</code>	Присваивает переменной <code>x</code> результат деления <code>x</code> на <code>y</code> . Если <code>x</code> и <code>y</code> – целые числа, остаток от деления теряется. Деление на ноль вызывает аварию ¹

Несмотря на то что в языке Go используются достаточно разумные правила определения старшинства операторов (в отличие, скажем, от C и C++), для большей очевидности рекомендуется использовать круглые скобки. Применение круглых скобок особенно рекомендуется программистам, использующим несколько языков программирования, чтобы избежать малозаметных ошибок.

¹ Авария – это исключение; см. главу 1 (выше) и §5.5 (ниже).

2.3.1. Целочисленные типы

В языке Go имеются 11 отдельных целочисленных типов, пять со знаком и пять без знака плюс один целочисленный тип для указателей – их имена и значения приводятся в табл. 2.5 (ниже). Кроме того, в Go допускается использовать имя `byte` как синоним беззнакового типа `uint8` и приветствуется использование имени `rune` как синонима типа `int32` при работе с отдельными символами (то есть кодовыми пунктами Юникода). Для большинства ситуаций достаточно использовать единственный целочисленный тип `int`. Переменные этого типа прекрасно подходят для использования в качестве счетчиков циклов, индексов массивов и срезов и арифметических вычислений общего назначения. Кроме того, этот тип обычно обеспечивает самую высокую скорость обработки. На момент написания этих строк тип `int` представлял 32-битные значения со знаком (даже на 64-битных платформах), но ожидается, что в будущих версиях Go он будет представлять 64-битные значения.

Другие целочисленные типы необходимы для организации чтения/записи целых чисел из внешних источников. Например, из файлов или из сетевых соединений. В таких случаях важно точно знать количество бит, которое следует прочитать или записать, чтобы обеспечить обработку целых чисел без искажений.

Таблица 2.5. Целочисленные типы и диапазоны представляемых значений

Тип	Диапазон представляемых значений
<code>byte</code>	Синоним типа <code>uint8</code>
<code>int</code>	Диапазон <code>int32</code> или <code>int64</code> , в зависимости от реализации
<code>int8</code>	[-128, 127]
<code>int16</code>	[-32768, 32767]
<code>int32</code>	[-2147483648, 2147483647]
<code>int64</code>	[-9223372036854775808, 9223372036854775807]
<code>rune</code>	Синоним типа <code>int32</code> .
<code>uint</code>	Диапазон <code>uint32</code> или <code>uint64</code> , в зависимости от реализации
<code>uint8</code>	[0, 255]
<code>uint16</code>	[0, 65535]
<code>uint32</code>	[0, 4294967295]
<code>uint64</code>	[0, 18446744073709551615]
<code>uintptr</code>	Беззнаковое целое, пригодное для хранения значения указателя

Таблица 2.6. Арифметические операторы, применимые только к встроенным целочисленным типам

Оператор	Описание/результат
$\sim x$	Поразрядное дополнение значения x
$x \% = y$	Присваивает переменной x остаток от деления x на y ; деление на нуль вызывает аварию
$x \& = y$	Присваивает переменной x результат поразрядной операции «И» над значениями x и y
$x = y$	Присваивает переменной x результат поразрядной операции «ИЛИ» над значениями x и y
$x \wedge = y$	Присваивает переменной x результат поразрядной операции «исключающее ИЛИ» над значениями x и y
$x \&\wedge = y$	Присваивает переменной x результат поразрядной операции «И-НЕ» над значениями x и y
$x >> = u$	Присваивает переменной x результат поразрядного сдвига вправо значения x на беззнаковое целое число u бит
$x << = u$	Присваивает переменной x результат поразрядного сдвига влево значения x на беззнаковое целое число u бит
$x \% y$	Остаток от деления x на y ; деление на нуль вызывает аварию
$x \& y$	Операция «поразрядное И»
$x y$	Операция «поразрядное ИЛИ»
$x \wedge y$	Операция «поразрядное исключающее ИЛИ»
$x \&\wedge y$	Операция «поразрядное И-НЕ»
$x << u$	Поразрядный сдвиг влево значения x на беззнаковое целое число u бит
$x >> u$	Поразрядный сдвиг вправо значения x на беззнаковое целое число u бит

Обычной практикой считается хранить целые числа в памяти в виде значений типа `int` и преобразовывать их в другие целочисленные типы со знаком и обратно при выполнении операций записи и чтения. Тип `byte` (`uint8`) используется для чтения и записи простых байтов, например при обработке текста в кодировке UTF-8. Простой пример чтения и записи текста в кодировке UTF-8 был представлен в предыдущей главе, в примере `americanise`. В главе 8 будет продемонстрирован другой пример, реализующий чтение и запись данных встроенных и пользовательских типов.

Для целочисленных значений в языке Go поддерживаются арифметические операции, перечисленные в табл. 2.4 (выше). Кроме того, к ним могут применяться арифметические и поразрядные операции,

перечисленные в табл. 2.6 (выше). Все эти операции обладают стандартным поведением, поэтому они не будут обсуждаться в дальнейшем, тем более что примеры их использования будут неоднократно встречаться на протяжении всей книги.

Целочисленное значение меньшего типа всегда можно преобразовать в значение большего типа без потери данных (например, из `int16` в `int32`); но обратное преобразование целого числа, слишком большого для целевого типа, или преобразование отрицательного значения в значение беззнакового типа приведет к потере данных или получению неожиданного значения без возбуждения аварийной ситуации. В таких случаях лучше использовать собственные функции преобразования в типы меньшего размера, такие как было показано выше (в разделе 2.3). Разумеется, при попытке преобразовать литерал со слишком большим значением (например, `int8(200)`) компилятор обнаружит проблему и сообщит об ошибке переполнения. Целые числа можно также преобразовывать в вещественные, используя стандартный прием преобразования типа (например, `float64(целое_число)`).

Поддержка 64-битных целых делает возможным в некоторых контекстах использовать масштабируемые целочисленные значения для выполнения точных вычислений. Например, использование в финансовых вычислениях значений типа `int64` для представления денежных сумм в миллионных долях центов позволяет оперировать суммами в миллиарды долларов с точностью, достаточной для большинства применений, особенно если с особой осторожностью подходить к операции деления. А если потребуется производить финансовые вычисления с высочайшей точностью и исключить ошибки округления, можно воспользоваться типом `big.Rat`.

2.3.1.1. Большие целые числа

В некоторых ситуациях требуется обеспечить высочайшую точность вычислений с целыми числами, диапазон которых превышает даже диапазон представления типов `int64` и `uint64`. В таких случаях нельзя использовать вещественные числа из-за ошибок округления. К счастью, в стандартной библиотеке Go имеются два безразмерных числовых типа: `big.Int` – для представления целых чисел со знаком, `big.Rat` – для представления рациональных значений (то есть чисел, которые могут быть представлены в виде дробей, например $\frac{2}{3}$ или 1.1496, но не в виде иррациональных значений, таких как e или π). Значения этих типов могут хранить числа, представленные любым количеством цифр,

ограничивая их размер только объемом доступной машинной памяти, но вычисления с подобными значениями выполняются намного медленнее, чем со значениями встроенных целочисленных типов.

Поскольку в языке Go, подобно языкам C и Java, отсутствует поддержка перегрузки операторов, для выполнения арифметических операций типы `big.Int` и `big.Rat` предоставляют методы с именами, такими как `Add()` и `Mul()`. В большинстве случаев методы изменяют значение своего приемника (то есть значения, относительно которого они вызываются), а также возвращают полученный результат в виде возвращаемого значения, чтобы обеспечить возможность составления цепочек вызовов методов. Здесь не будут перечисляться все функции и методы, предоставляемые пакетом `math/big`, поскольку их легко можно найти в документации, а также потому, что к моменту выхода книги могли появиться новые. Однако ниже будет представлен достаточно полный пример, демонстрирующий особенности использования значений типа `big.Int`.

Тип `float64` позволяет производить точные вычисления со значениями, насчитывающими до 15 десятичных знаков, чего более чем достаточно в большинстве ситуаций. Однако если потребуется вычислить большее количество десятичных знаков, скажем, несколько десятков или даже сотен, как, например, при вычислении числа π , встроенных типов будет недостаточно.

В 1706 году Джон Мэчин (John Machin) вывел формулу для вычисления числа π с произвольной точностью, которую можно адаптировать для реализации на языке Go вычислений числа π с произвольной точностью с использованием значений типа `big.Int`. Сама формула и функция `arccot()`, на которую она опирается, представлены на рис. 2.1. (Чтобы использовать тип `big.Int`, не обязательно понимать формулу Мэчина.) Наша реализация функции `arccot()` принимает дополнительный аргумент, ограничивающий точность вычислений, чтобы не превысить требуемую точность вычислений.

Вся программа занимает менее 80 строк и находится в файле `pi_by_digits/pi_by_digits.go`; ниже приводится ее функция `main()`¹.

```
func main() {
    places := handleCommandLine(1000)
    scaledPi := fmt.Sprintf( $\pi$ (places))
    fmt.Printf("3.%s\n", scaledPi[1:])
}
```

¹ Реализация, представленная здесь, основана на реализации [http://en.literateprograms.org/Pi_with_Machin's_formula_\(Python\)](http://en.literateprograms.org/Pi_with_Machin's_formula_(Python)).

$$\pi = 4(4\operatorname{arccot}(5) - \operatorname{arccot}(239)) \quad \operatorname{arccot}(x) = \frac{1}{x} - \frac{1}{3x^3} + \frac{1}{5x^5} - \frac{1}{7x^7} + \dots$$

Рис. 2.1. Формула Мэчина

По умолчанию программа вычисляет число π с точностью до 1000 десятичных знаков, однако пользователь может выбрать любое другое число, передав его в виде аргумента командной строки. Функция `handleCommandLine()` (здесь не показана) возвращает значение, переданное ей, или число, указанное пользователем в командной строке (если оно имеется и допустимо). Функция `л()` возвращает число π в виде значения `314159...` типа `big.Int`; это значение преобразуется в строку и затем выводится в консоль в отформатированном виде, чтобы оно имело вид: 3.14159265358979323846264338327950288419716939937510 (здесь число π вычислено с точностью до 50 знака).

```
func л(places int) *big.Int {
    digits := big.NewInt(int64(places))
    unity := big.NewInt(0)
    ten := big.NewInt(10)
    exponent := big.NewInt(0)
    unity.Exp(ten, exponent.Add(digits, ten), nil) ❶
    pi := big.NewInt(4)
    left := arccot(big.NewInt(5), unity)
    left.Mul(left, big.NewInt(4)) ❷
    right := arccot(big.NewInt(239), unity)
    left.Sub(left, right)
    pi.Mul(pi, left) ❸
    return pi.Div(pi, big.NewInt(0).Exp(ten, ten, nil)) ❹
}
```

Функция `л()` начинается с вычисления значения переменной `unity` (`10знаков+10`), которое используется как коэффициент масштабирования для вычислений с использованием целых чисел. Слагаемое `+10` увеличивает на десять число цифр, указанное пользователем, чтобы избежать ошибок округления. Затем используется формула Мэчина с модифицированной версией функции `arccot()` (здесь не показана), которой во втором аргументе передается переменная `unity`. В конце возвращается результат, деленный на `1010`, чтобы обратить эффект увеличения коэффициента масштабирования `unity`.

Чтобы присвоить переменной `unity` требуемое значение, создаются четыре переменные, все типа `*big.Int` (то есть указатели на значение типа `big.Int`; см. §4.1). Переменные `unity` и `exponent` инициализируются значением 0, переменная `ten` – значением 10, а переменная `digits` – числом цифр, полученным от пользователя. Вычисление значения переменной `unity` производится в одной строке (❶). Метод `big.Int.Add()` увеличивает число цифр на 10. Затем метод `big.Int.Exp()` возводит число 10 в степень, определяемую вторым аргументом (`digits + 10`). Когда в третьем аргументе передается значение `nil`, как в данной реализации, метод `big.Int.Exp(x, y, nil)` вычисляет значение выражения x^y ; если в третьем аргументе передать непустое значение, метод `big.Int.Exp(x, y, z)` вычислит значение выражения $x^y \bmod z$. Обратите внимание, что здесь нет необходимости присваивать полученное значение переменной `unity`, потому что большинство методов типа `big.Int` не только возвращают результат, но и сохраняют его в своем приемнике, поэтому здесь переменная `unity` автоматически получит вычисленное значение.

Остальные вычисления производятся по той же схеме. Переменной `pi` присваивается начальное значение 4, и затем вычисляется значение левого операнда во внутренней части формулы Мэчина. Здесь не требуется присваивать полученное значение переменной `left` (❷), потому что метод `big.Int.Mul()` не только вернет полученное значение (которое можно просто игнорировать), но и сохранит его в своем приемнике (то есть в данном случае в переменной `left`). Затем вычисляется значение правого операнда во внутренней части формулы и из значения переменной `left` вычитается значение переменной `right` (результат автоматически сохраняется в переменной `left`). Далее значение переменной `pi` (4) умножается на значение переменной `left` (результат вычислений по формуле Мэчина). В итоге получается значение, масштабированное увеличенным значением переменной `unity`. Поэтому в последней строке (❸) выполняется возврат к требуемому масштабу путем деления результата (в `pi`) на 10^{10} .

Использование значений типа `big.Int` требует определенной внимательности, потому что большинство методов этого типа изменяют значение своего приемника (это сделано для эффективности, чтобы сэкономить на создании временных значений типа `big.Int`). Сравните строку, где вычисляется выражение `pi * left` и результат сохраняется в переменной `pi` (❹ в листинге выше), со строкой, где вычисляется выражение `pi ÷ 1010` и результат передается инструкции

return (4 в листинге выше) – здесь уже не важно, что переменной pi будет присвоено значение результата вычислений.

Всегда, когда это возможно, следует использовать простой тип int, переходя на использование типа int64, если диапазон значений типа int окажется слишком мал, или на использование типов float32 и float64, если ошибками округления можно пренебречь. Однако в вычислениях, требующих высокой точности, когда цена точности оказывается выше цены используемой памяти и скорости вычислений, можно использовать значения типов big.Int и big.Rat – последний особенно удобно применять в финансовых вычислениях, когда требуется оперировать вещественными значениями, используя прием масштабирования, как в данном примере.

2.3.2. Вещественные типы

В языке Go имеется два типа для представления вещественных чисел и два типа для представления комплексных чисел – их имена и диапазоны представлены в табл. 2.7. Вещественные числа в языке Go хранятся в широко применяемом формате IEEE-754 (http://ru.wikipedia.org/wiki/IEEE_754-2008). Этот формат используется многими микропроцессорами и арифметическими сопроцессорами, поэтому в большинстве случаев программы на языке Go могут пользоваться преимуществами аппаратной поддержки арифметики с вещественными числами.

Таблица 2.7. Вещественные типы в языке Go

Тип	Диапазон
float32	$\pm 3,402823466385288598117041834 \times 10^{38}$ Мантисса может быть представлена с точностью до 7 десятичных разрядов
float64	$\pm 1,797693134 \times 10^{308}$ Мантисса может быть представлена с точностью до 15 десятичных разрядов
complex64	Обе части, действительная и мнимая, представлены значениями типа float32
complex128	Обе части, действительная и мнимая, представлены значениями типа float64

В языке Go поддерживаются все операции с вещественными числами, перечисленные в табл. 2.4 (выше). Большинство констант из пакета math, и все его функции перечислены в табл. 2.8–2.10 (ниже).

Таблица 2.8. Константы и функции из пакета *math* (часть 1)

Все функции из пакета *math* принимают и возвращают значения типа *float64*, если явно не указано иное. Значения всех показанных здесь констант округлены до 15 десятичных знаков, чтобы уместить их в ячейки таблицы.

Функция/константа	Описание/результат
<code>math.Abs(x)</code>	$ x $, абсолютное значение x
<code>math.Acos(x)</code>	Арккосинус угла x , выраженного в радианах
<code>math.Acosh(x)</code>	Гиперболический арккосинус угла x , выраженного в радианах
<code>math.Asin(x)</code>	Арсинус угла x , выраженного в радианах
<code>math.Asinh(x)</code>	Гиперболический арксинус угла x , выраженного в радианах
<code>math.Atan(x)</code>	Артангенс угла x , выраженного в радианах
<code>math.Atan2(y, x)</code>	Артангенс угла y/x , выраженного в радианах
<code>math.Atanh(x)</code>	Гиперболический арктангенс угла x , выраженного в радианах
<code>math.Cbrt(x)</code>	$\sqrt[3]{x}$, кубический корень из x
<code>math.Ceil(x)</code>	$\lceil x \rceil$, наименьшее целое, большее или равное x ; например <code>math.Ceil(5.4) == 6.0</code>
<code>math.Copysign(x, y)</code>	Результат имеет значение x и знак y
<code>math.Cos(x)</code>	Косинус угла x , выраженного в радианах
<code>math.Cosh(x)</code>	Гиперболический косинус угла x , выраженного в радианах
<code>math.Dim(x, y)</code>	Фактически <code>math.Max(x - y, 0.0)</code>
<code>math.E</code>	Константа e ; примерно 2.718281828459045
<code>math.Erf(x)</code>	$\text{erf}(x)$; гауссова функция ошибки от значения x
<code>math.Erfc(x)</code>	$\text{erfc}(x)$; дополнительная гауссова функция ошибки от значения x
<code>math.Exp(x)</code>	e^x
<code>math.Exp2(x)</code>	2^x
<code>math.Expm1(x)</code>	$e^x - 1$, дает более точный результат, чем выражение <code>Exp(x) - 1</code> , когда значение x близко к 0
<code>math.Float32bits(f)</code>	Двоичное представление в формате IEEE-754 числа f (типа <code>float32</code>) в виде <code>uint32</code>
<code>math.Float32frombits(u)</code>	Двоичное представление в формате IEEE-754 битов u (типа <code>uint32</code>) в виде значения вещественного типа <code>float32</code>
<code>math.Float64bits(x)</code>	Двоичное представление в формате IEEE-754 числа x (типа <code>float64</code>) в виде <code>uint64</code>
<code>math.Float64frombits(u)</code>	Двоичное представление в формате IEEE-754 битов u (типа <code>uint64</code>) в виде значения вещественного типа <code>float64</code>

Таблица 2.9. Константы и функции из пакета *math* (часть 2)

Функция/константа	Описание/результат
<code>math.Floor(x)</code>	$\lfloor x \rfloor$, наибольшее целое, меньшее или равное x ; например <code>math.Floor(5.4) == 5.0</code>
<code>math.Frexp(x)</code>	Дробная часть <code>frac</code> типа <code>float64</code> и порядок <code>exp</code> типа <code>int</code> такие, что $x = \text{frac} \times 2^{\text{exp}}$. Обратная функции <code>math.Ldexp()</code>
<code>math.Gamma(x)</code>	$\Gamma(x)$, то есть $(x - 1)!$
<code>math.Hypot(x, y)</code>	<code>math.Sqrt(x * x, y * y)</code>
<code>math.Ilogb(x)</code>	Порядок двойки для x в виде числа типа <code>int</code> ; см. также <code>math.Logb()</code>
<code>math.Inf(n)</code>	Значение типа <code>float64</code> , соответствующее значению $+\infty$, если n типа <code>int</code> больше или равно 0; и $-\infty$ – в противном случае
<code>math.IsInf(x, n)</code>	Возвращает <code>true</code> , если x типа <code>float64</code> является $+\infty$ и n типа <code>int</code> больше 0, или если x является $-\infty$ и n меньше 0, или если x является любой из бесконечностей и n равно 0; в противном случае возвращает <code>false</code>
<code>math.IsNaN(x)</code>	Возвращает <code>true</code> , если x имеет значение «не число» в формате IEEE-754
<code>math.J0(x)</code>	$J_0(x)$, функция Бесселя первого рода
<code>math.J1(x)</code>	$J_1(x)$, функция Бесселя первого рода
<code>math.Jn(n, x)</code>	$J_n(x)$, функция Бесселя первого рода n -го порядка (где n имеет тип <code>int</code>)
<code>math.Ldexp(x, n)</code>	$x \times 2^n$, где x имеет тип <code>float64</code> , а n – тип <code>int</code> ; обратная функции <code>math.Frexp()</code>
<code>math.Lgamma(x)</code>	$\log_e(\Gamma(x))$, результат имеет тип <code>float64</code> , а знак $\Gamma(x)$, как значение -1 или $+1$ типа <code>int</code>
<code>math.Ln2</code>	$\log_e(2)$, примерно 0,693147180559945
<code>math.Ln10</code>	$\log_{10}(2)$, примерно 2,302585092994045
<code>math.Log(x)</code>	$\log_e(x)$
<code>math.Log2E</code>	$1/\log_e(2)$, примерно 1,442695021629333
<code>math.Log10(x)</code>	$\log_{10}(x)$
<code>math.Log10E</code>	$1/\log_e(10)$, примерно 0,434294492006301
<code>math.Log1p(x)</code>	$\log_e(1 + x)$, дает более высокую точность при значениях x , близких к нулю, чем <code>math.Log()</code>
<code>math.Log2(x)</code>	$\log_2(x)$
<code>math.Logb(x)</code>	Порядок двойки для x ; см. также <code>math.Ilogb()</code>
<code>math.Max(x, y)</code>	Наибольшее из чисел x и y
<code>math.Min(x, y)</code>	Наименьшее из чисел x и y
<code>math.Mod(x, y)</code>	Остаток от деления x на y ; см. также <code>math.Remainder()</code>

Таблица 2.10. Константы и функции из пакета *math* (часть 3)

Функция/константа	Описание/результат
<code>math.Modf(x)</code>	Целая и дробная части числа x в виде значений типа <code>float64</code>
<code>math.NaN(x)</code>	Значение «не число» в формате IEEE-754
<code>math.Nextafter(x, y)</code>	Следующее представимое значение, следующее после x в направлении к y
<code>math.Pi</code>	Константа π , примерно 3,141592653589793
<code>math.Phi</code>	Константа ϕ ; примерно 1,618033988749984
<code>math.Pow(x, y)</code>	x^y
<code>math.Pow10(n)</code>	10^n в виде значения типа <code>float64</code> , где значение n имеет тип <code>int</code>
<code>math.Remainder(x, y)</code>	IEEE-754-совместимый остаток от деления x на y , см. также <code>math.Mod()</code>
<code>math.Signbit(x)</code>	Возвращает значение типа <code>bool</code> ; <code>true</code> , если x является отрицательным числом (включая -0.0)
<code>math.Sin(x)</code>	Синус угла x , выраженного в радианах
<code>math.SinCos(x)</code>	Синус и косинус угла x , выраженного в радианах
<code>math.Sinh(x)</code>	Гиперболический синус угла x , выраженного в радианах
<code>math.Sqrt(x)</code>	\sqrt{x}
<code>math.Sqrt2</code>	$\sqrt{2}$, примерно 1,414213562373095
<code>math.SqrtE</code>	\sqrt{e} , примерно 1,648721270700128
<code>math.SqrtPi</code>	$\sqrt{\pi}$, примерно 1,772453850905516
<code>math.SqrtPhi</code>	$\sqrt{\phi}$, примерно 1,272019649514068
<code>math.Tan(x)</code>	Тангенс угла x , выраженного в радианах
<code>math.Tanh(x)</code>	Гиперболический тангенс угла x , выраженного в радианах
<code>math.Trunc(x)</code>	x с усеченной дробной частью
<code>math.Y0(x)</code>	$Y_0(x)$, функция Бесселя второго рода
<code>math.Y1(x)</code>	$Y_1(x)$, функция Бесселя второго рода
<code>math.Yn(n, x)</code>	$Y_n(x)$, функция Бесселя второго рода n -го порядка (где n имеет тип <code>int</code>)

Вещественные числа записываются в форме с десятичной точкой или в экспоненциальной форме, например 0.0, 3., 8.2, -7.4, -6e4, .1, 5.9E-3. В компьютерах для внутреннего представления вещественных чисел используется двоичная форма, то есть некоторые

вещественные числа могут быть представлены точно (такие как 0.5), а другие – только приблизительно (такие как 0.1 и 0.2). Кроме того, для внутреннего представления используется фиксированное число бит, что ограничивает число цифр в десятичном представлении. Эта проблема характерна не только для языка Go, но и для всех основных языков программирования. Однако неточность вещественных чисел не всегда очевидна, потому что в языке Go используется весьма интеллектуальный алгоритм вывода вещественных чисел, который выводит минимально возможное число цифр, необходимых для обеспечения поддерживаемой точности.

С вещественными числами могут использоваться все операторы сравнения, перечисленные в табл. Table 2.3 (выше). К сожалению, из-за того, что вещественные числа хранят лишь приблизительные значения, сравнение на равенство или неравенство не всегда дает ожидаемые результаты.

```
x, y := 0.0, 0.0
for i := 0; i < 10; i++ {
    x += 0.1
    if i%2 == 0 {
        y += 0.2
    } else {
        fmt.Printf("%-5t %-5t %-5t %-5t", x == y,
            EqualFloat(x, y, -1), EqualFloat(x, y, 0.0000000000001),
            EqualFloatPrec(x, y, 6))
        fmt.Println(x, y)
    }
}
```

true	true	true	true	0.2	0.2
true	true	true	true	0.4	0.4
false	false	true	true	0.6	0.60000000000000001
false	false	true	true	0.7999999999999999	0.8
false	false	true	true	0.9999999999999999	1

Здесь сначала создаются две переменные типа `float64` с начальными значениями, равными 0. Затем к первой из них десять раз добавляется значение 0.1, ко второй – пять раз значение 0.2, то есть в результате обе переменные должны получить значение 1. Однако, как следует из вывода, представленного ниже, для некоторых вещественных значений невозможно достичь полной точности представления. Ввиду этого необходимо проявлять особую осторожность при сравнении вещественных чисел между собой с помощью операторов

`==` и `!=`. Однако бывают ситуации, когда сравнение вещественных чисел на равенство или неравенство с помощью встроенных операторов имеет особый смысл, например чтобы избежать деления на ноль: `if y != 0.0 { return x / y }`.

Спецификатор формата `"%-5t"` выводит логическое значение с выравниванием по левому краю в поле шириной пять символов (форматирование строк подробно рассматривается в следующей главе, в §3.5).

```
func EqualFloat(x, y, limit float64) bool {
    if limit <= 0.0 {
        limit = math.SmallestNonzeroFloat64
    }
    return math.Abs(x-y) <=
        (limit * math.Min(math.Abs(x), math.Abs(y)))
}
```

Функция `EqualFloat()` сравнивает два значения типа `float64` с заданной точностью или с самой большой аппаратной точностью, которой можно достичь, если в аргументе `limit` было передано отрицательное число (например, `-1`). Она опирается на использование функций (и констант) из пакета `math`, входящего в состав стандартной библиотеки.

Можно также использовать другой (более медленный) способ сравнения чисел в виде строк.

```
func EqualFloatPrec(x, y float64, decimals int) bool {
    a := fmt.Sprintf("%.*f", decimals, x)
    b := fmt.Sprintf("%.*f", decimals, y)
    return len(a) == len(b) && a == b
}
```

Точность для этой функции определяется количеством знаков после десятичной точки. В спецификаторе формата `%`, в функции `fmt.Sprintf()`, можно использовать шаблонный символ `*`, на место которого будет подставляться число, то есть в данном примере создаются две строки для заданных значений типа `float64`, для каждого из которых указывается определенное число десятичных знаков. Если числа будут существенно отличаться по величине, соответственно, будут отличаться и длины строк `a` и `b` (например, `12.32` и `592.85`), поэтому сравнение таких чисел будет выполняться быстрее. (Форматирование строк подробно рассматривается в §3.5.)

В большинстве случаев, где необходимы вещественные числа, лучше использовать значения типа `float64`, особенно если учесть, что все функции в пакете `math` оперируют значениями типа `float64`. Однако в языке Go имеется также тип `float32`, который может пригодиться, когда на первое место выходит проблема экономии памяти и не требуется применение функций из пакета `math` или когда неудобство преобразования значений в тип `float64` и обратно не кажется таким значительным. Поскольку в языке Go вещественные числа имеют строго определенный размер, их всегда можно без опаски читать из внешних источников, таких как файлы и сетевые соединения, или записывать в них.

Вещественные числа можно преобразовывать в целые значения, используя стандартный синтаксис (например, `int(вещественное_число)`), в этом случае дробная часть просто отбрасывается. Разумеется, если вещественное число превосходит по величине максимально возможное значение целочисленного типа, в который выполняется преобразование, в результате будет получено непредсказуемое значение. Решить эту проблему можно с помощью собственной функции преобразования, например:

```
func IntFromFloat64(x float64) int {
    if math.MinInt32 <= x && x <= math.MaxInt32 {
        whole, fraction := math.Modf(x)
        if fraction >= 0.5 {
            whole++
        }
        return int(whole)
    }
    panic(fmt.Sprintf("%g is out of the int32 range", x))
}
```

В спецификации языка Go (golang.org/doc/go_spec.html) отмечается, что значение типа `int` занимает то же количество бит, что и значение типа `uint`, а значение типа `uint` всегда занимает 32 или 64 бита. Отсюда следует, что значение типа `int` занимает по меньшей мере 32 бита, то есть можно без опаски использовать константы `math.MinInt32` и `math.MaxInt32` для определения граничных значений, представляемых типом `int`.

Функция `math.Modf()` в этом примере используется для выделения дробной и целой частей числа (обе имеют тип `float64`), но, вместо того чтобы просто вернуть целую часть (то есть выполнить

усечение), функция в примере выше округляет целую часть в большую сторону, если дробная часть больше или равна 0.5.

Вместо того чтобы просто вернуть ошибку, как это делает функция `Uint8FromInt()` (в §2.3), в этой функции выход значения за допустимый диапазон считается веской причиной, чтобы прервать выполнение программы. Поэтому здесь была использована встроенная функция `panic()`, вызывающая аварию во время выполнения и останавливающая выполнение программы, если аварийная ситуация не будет обработана вызовом функции `recover()` (§5.5). То есть, если программа была выполнена успешно, можно быть уверенными, что она не пыталась преобразовать значение, выходящее за допустимый диапазон. (Обратите также внимание на отсутствие инструкции `return` в конце функции – компилятор Go понимает, что в случае вызова функции `panic()` невозможно нормальное завершение функции в этой точке.)

2.3.2.1. Комплексные типы

Как видно в табл. Table 2.7 (выше), в языке Go поддерживаются два типа комплексных чисел. Комплексные числа могут создаваться с помощью встроенной функции `complex()` или с использованием литералов, включающих мнимую часть. Компоненты комплексных чисел можно извлекать встроенными функциями `real()` и `imag()`, каждая из которых возвращает значение типа `float64` (или `float32` – для значений типа `complex64`).

Для комплексных чисел поддерживаются все арифметические операции, перечисленные в табл. 2.4 (выше). Из операторов сравнения к комплексным числам могут применяться только операторы `==` и `!=` (см. табл. 2.3 выше), но операциям сравнения комплексных чисел свойственны те же проблемы, что и операциям сравнения вещественных чисел. Для работы с комплексными числами в стандартной библиотеке имеется специализированный пакет `math/cmplx`, который содержит функции, перечисленные в табл. 2.11.

Ниже приводятся несколько простых примеров комплексных чисел:

```
f := 3.2e5           // тип: float64
x := -7.3 - 8.9i     // тип: complex128 (literal)
y := complex64(-18.3 + 8.9i) // тип: complex64 (conversion) ❶
z := complex(f, 13.2) // тип: complex128 (construction) ❷
fmt.Println(x, real(y), imag(z)) // Выведет: (-7.3-8.9i) -18.3 13.2
```

В языке Go мнимая часть комплексного числа обозначается с помощью окончания `i`, как это принято в математике¹. В этом примере переменные `x` и `z` имеют тип `complex128`, то есть их действительные и мнимые части являются значениями типа `float64`. Переменная `y` имеет тип `complex64`, а компоненты ее значения – тип `float32`. Важно отметить, что использование имени типа `complex64` (как и любого другого) наподобие функции выполняет операцию преобразования типа. Поэтому здесь (❶) комплексное число `-18.3+8.9i` (типа `complex128` – автоматически определяется компилятором для литералов комплексных чисел) преобразуется в тип `complex64`. Однако инструкция `complex()` – это функция (в Go нет типа с таким именем), принимающая два вещественных числа и возвращающая соответствующее им комплексное число типа `complex128` (❷).

Еще одна тонкость заключается в том, что функция `fmt.Println()` может выводить комплексные числа без лишних формальностей. (Как будет показано в главе 6, имеется возможность организовать бесшовное взаимодействие пользовательских типов с функциями вывода в языке Go, просто снабдив их методом `String()`.)

В общем случае из всех типов комплексных чисел лучше использовать тип `complex128`, потому что все функции в пакете `math/cmplx` оперируют значениями типа `complex128`. Однако в Go имеется также тип `complex64`, который может пригодиться для экономного расходования памяти. Поскольку в языке Go значения комплексных чисел имеют строго определенный размер, их всегда можно без опаски читать из внешних источников, таких как файлы и сетевые соединения, или записывать в них.

В этой главе были представлены логические и числовые типы данных, а также таблицы с операторами и функциями, предназначенными для работы с этими типами. В следующей главе рассматривается тип `string`, включая подробное описание возможностей форматирования строк в функциях вывода (§3.5), в том числе форматирование логических и числовых значений. Особенности чтения и записи в файлы данных различных типов, в том числе логические и числовые значения, описываются в главе 8. Однако, прежде чем закончить эту главу, рассмотрим небольшой, но законченный пример действующей программы.

¹ В противоположность техническим областям знаний и языку Python, где для обозначения мнимой части числа используется символ `j`.

Таблица 2.11. Функции для работы с комплексными числами в пакете *math*

Функция/кон-станта	Описание/результат
<code>cmplx.Abs(x)</code>	$ x $, абсолютное значение x в виде значения типа <code>float64</code>
<code>cmplx.Acos(x)</code>	Арккосинус угла x , выраженного в радианах
<code>cmplx.Acosh(x)</code>	Гиперболический арккосинус угла x , выраженного в радианах
<code>cmplx.Asin(x)</code>	Арсинус угла x , выраженного в радианах
<code>cmplx.Asinh(x)</code>	Гиперболический арксинус угла x , выраженного в радианах
<code>cmplx.Atan(x)</code>	Арктангенс угла x , выраженного в радианах
<code>cmplx.Atanh(x)</code>	Гиперболический арктангенс угла x , выраженного в радианах
<code>cmplx.Conj(x)</code>	Комплексно-сопряженное число дл числа x
<code>cmplx.Cos(x)</code>	Косинус угла x , выраженного в радианах
<code>cmplx.Cosh(x)</code>	Гиперболический косинус угла x , выраженного в радианах
<code>cmplx.Cot(x)</code>	Котангенс угла x , выраженного в радианах
<code>cmplx.Exp(x)</code>	e^x
<code>cmplx.Inf()</code>	<code>complex(math.Inf(1), math.Inf(1))</code>
<code>cmplx.IsInf(x)</code>	<code>true</code> , если <code>real(x)</code> или <code>imag(x)</code> вернет $\pm\infty$, иначе <code>false</code>
<code>cmplx.IsNaN(x)</code>	<code>true</code> , если <code>real(x)</code> или <code>imag(x)</code> вернет значение «не число» и ни одна из них не вернет $\pm\infty$, иначе <code>false</code>
<code>cmplx.Log(x)</code>	$\log_e(x)$
<code>cmplx.Log10(x)</code>	$\log_{10}(x)$
<code>cmplx.NaN()</code>	Комплексное значение «не число»
<code>cmplx.Phase(x)</code>	Фаза числа x в виде значения типа <code>float64</code> в диапазоне $[-\pi, +\pi]$
<code>cmplx.Polar(x)</code>	Абсолютное значение r и фаза Θ , оба числа типа <code>float64</code> , удовлетворяющие условию $x = r \times e^{i\Theta}$, где фаза находится в диапазоне $[-\pi, +\pi]$
<code>cmplx.Pow(x, y)</code>	x^y
<code>cmplx.Rect(r, q)</code>	Значение типа <code>complex128</code> с полярными координатами r и q типа <code>float64</code>
<code>cmplx.Sin(x)</code>	Синус угла x , выраженного в радианах
<code>cmplx.Sinh(x)</code>	Гиперболический синус угла x , выраженного в радианах
<code>cmplx.Sqrt(x)</code>	\sqrt{x}
<code>cmplx.Tan(x)</code>	Тангенс угла x , выраженного в радианах
<code>cmplx.Tanh(x)</code>	Гиперболический тангенс угла x , выраженного в радианах

2.4. Пример: statistics

Цель этого примера (и всех последующих) – дать общее представление (и практические навыки) о программировании на языке Go. Как и в главе 1, в обсуждаемом примере используются некоторые особенности языка Go, которые еще не рассматривались во всех подробностях. Однако это не должно вызывать проблем, потому что пример будут сопровождать краткие пояснения и ссылки на соответствующие главы. Пример также знакомит с некоторыми простейшими приемами использования пакета `net/http` из стандартной библиотеки Go – этот пакет значительно упрощает создание HTTP-серверов. В соответствии с основной темой главы пример и последующие упражнения связаны с обработкой чисел.

Программа `statistics` (в файле `statistics/statistics.go`) – это *веб-приложение*, предлагающее пользователю ввести массив чисел и выполняющее простейшие статистические вычисления. На рис. 2.2 показана программа в действии. Обзор программного кода будет выполнен в два этапа, сначала будет рассматриваться реализация математической функциональности, а затем реализация веб-страницы приложения. Здесь не будет демонстрироваться весь программный код приложения (например, в листингах отсутствуют инструкции импортирования и объявления большинства констант), поскольку он доступен для загрузки, но достаточно большая его часть, чтобы было понятно, как действует приложение.

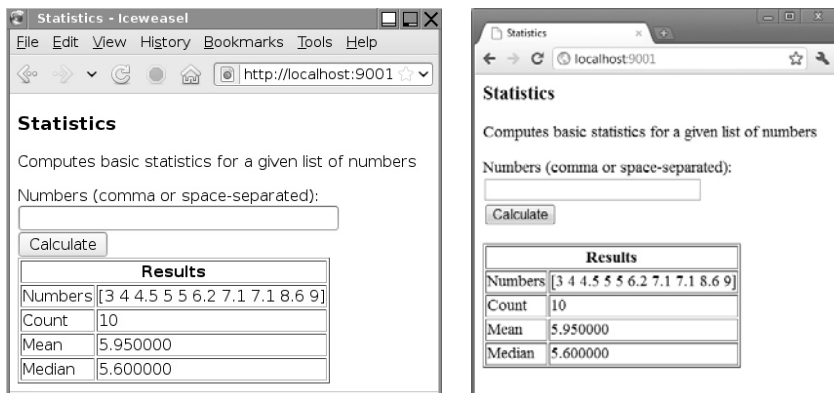


Рис. 2.2. Программа `statistics` в Linux и Windows

2.4.1. Реализация простых статистических функций

Для удобства в программе определяется составной тип для хранения чисел, введенных пользователем, и двух статистических значений, которые планируется вычислить.

```
type statistics struct {  
    numbers []float64  
    mean    float64  
    median  float64  
}
```

Структуры в языке Go похожи на структуры в языке C и на классы в языке Java, содержащие только общедоступные поля и не имеющие методов, но отличаются от структур в языке C++, поскольку не являются замаскированными классами. Как будет показано далее, структуры в Go обеспечивают отличную поддержку агрегирования и встраивания (§6.4), а также часто образуют ядро объектно-ориентированной функциональности в программах на языке Go (глава 6).

```
func getStats(numbers []float64) (stats statistics) {  
    stats.numbers = numbers  
    sort.Float64s(&stats.numbers)  
    stats.mean = sum(numbers) / float64(len(numbers))  
    stats.median = median(numbers)  
    return stats  
}
```

Эта функция принимает срез с числами (в данном случае полученных функцией `processRequest()` ниже) и заполняет возвращаемое значение `stats` (типа `statistics`) результатами вычислений. Для вычисления медианы необходимо отсортировать последовательность чисел в порядке возрастания, что реализуется с помощью функции `Float64s()` из пакета `sort`, которая сортирует срезы типа `[]float64` на месте. Это означает, что функция `getStats()` модифицирует свои аргументы, что часто происходит, когда функциям передаются срезы, ссылки или указатели. Если бы в программе потребовалось сохранить оригинальный порядок следования чисел в срезе, можно было бы создать временную копию среза с помощью встроенной функции `copy()` (§4.2.3) и работать с копией.

Математическое ожидание (или среднее) – это простая сумма всех значений в последовательности, деленная на количество этих значений. Сумма здесь вычисляется с помощью отдельной вспомогательной функции, а количество значений (длина среза) преобразуется в значение типа `float64`, чтобы обеспечить совместимость типов (так как `sum()` возвращает значение типа `float64`). Это гарантирует выполнение вещественного деления и отсутствие усечения результата, неизбежного при использовании целых чисел. *Медиана* – это срединное значение, оно вычисляется отдельно, с помощью функции `median()`.

Здесь отсутствует проверка деления на ноль, поскольку сама логика программы подразумевает, что `getStats()` будет вызываться, только когда имеется хотя бы одно число, поэтому, если в будущем логика работы программы изменится, она будет завершаться аварийно в таких ситуациях. В критически важных программах, которые не должны завершаться при возникновении проблем, можно задействовать функцию `recover()`, чтобы восстанавливать приложение в нормальное состояние после аварий и продолжать работу (§5.5).

```
func sum(numbers []float64) (total float64) {
    for _, x := range numbers {
        total += x
    }
    return total
}
```

Для обхода всех чисел и вычисления суммы эта функция использует цикл `for ...range` (отбрасывающий значения их индексов). Благодаря тому что в языке Go переменные, включая именованные возвращаемые значения, всегда *инициализируются нулевыми значениями*, значение `total` изначально равно нулю.

```
func median(numbers []float64) float64 {
    middle := len(numbers) / 2
    result := numbers[middle]
    if len(numbers)%2 == 0 {
        result = (result + numbers[middle-1]) / 2
    }
    return result
}
```

Этой функции должен передаваться отсортированный срез со значениями типа `float64`. Сначала она принимает в качестве медианы

значение из середины среза, но если срез содержит четное количество элементов, в середине фактически оказываются два значения, и в этом случае вычисляется среднее арифметическое двух срединных значений. И в конце результат возвращается вызывающей программе.

В этом подразделе был описан порядок обработки данных в приложении. В следующем подразделе мы рассмотрим основы реализации инфраструктуры поддержки веб-приложения, состоящего из единственной веб-страницы. (Читатели, кому не интересна тема веб-программирования, могут сразу перейти к упражнениям или к следующей главе.)

2.4.2. Реализация простого HTTP-сервера

Программа `statistics` предоставляет доступ к единственной веб-странице на локальном компьютере. Ниже приводится функция `main()` программы:

```
func main() {
    http.HandleFunc("/", homePage)
    if err := http.ListenAndServe(":9001", nil); err != nil {
        log.Fatal("failed to start server", err)
    }
}
```

Функция `http.HandleFunc()` принимает два аргумента: путь и ссылку на функцию, которую следует вызвать при поступлении запроса по указанному пути. Функция должна иметь сигнатуру `func(http.ResponseWriter, *http.Request)`. Приложение может зарегистрировать неограниченное количество пар путь/функция. Здесь регистрируются путь `/` (то есть домашняя страница веб-приложения) и пользовательская функция `homePage()`.

Функция `http.ListenAndServe()` запускает веб-сервер, принимающий запросы на указанном сетевом TCP-адресе, — здесь используется локальный адрес компьютера и порт с номером 9001. Когда указывается только номер порта, автоматически предполагается, что адрес соответствует локальному компьютеру — с тем же успехом можно было бы использовать адрес `"localhost:9001"` или `"127.0.0.1:9001"`. (Номер порта для данного приложения был выбран совершенно произвольно, вместо него можно указать другой, если использование этого номера вызывает конфликты с существующим сервером.) Второй аргумент определяет тип сервера. Обычно в нем передается `nil`, чтобы выбрать тип по умолчанию.

В программе определены несколько строковых констант, но здесь показана только одна из них.

```
form = `<form action="/" method="POST">
<label for="numbers">Numbers (comma or space-separated):</label><br />
<input type="text" name="numbers" size="30"><br />
<input type="submit" value="Calculate">
</form>`
```

Строковая константа `form` содержит элемент `<form>`, включающий элементы `<input>` типа `text` и `submit`.

```
func homePage(writer http.ResponseWriter, request *http.Request) {
    err := request.ParseForm() // Должна вызываться перед записью в ответ
    fmt.Fprint(writer, pageTop, form)
    if err != nil {
        fmt.Fprintf(writer, anError, err)
    } else {
        if numbers, message, ok := processRequest(request); ok {
            stats := getStats(numbers)
            fmt.Fprint(writer, formatStats(stats))
        } else if message != "" {
            fmt.Fprintf(writer, anError, message)
        }
    }
    fmt.Fprint(writer, pageBottom)
}
```

Эта функция вызывается при каждом посещении веб-сайта приложения `statistics`. В аргументе `writer` передается значение, куда должен записываться ответ (в формате HTML), а в аргументе `request` — подробная информация о запросе.

Функция начинается с анализа формы (которая изначально имеет пустой текстовый элемент `<input>`). Текстовому элементу `<input>` присвоено имя «`numbers`», чтобы на него можно было ссылаться позднее, при обработке формы. Кроме того, атрибуту `action` формы присвоено значение `/`, чтобы после щелчка на кнопке **Calculate** (Рассчитать) браузер запрашивал ту же самую страницу. Это означает, что функция `homePage()` будет вызываться для обработки всех запросов, поэтому она должна уметь обрабатывать первичный вызов, когда еще не было введено ни одного числа, и последующие вызовы, когда числа были введены или когда возникла ошибка. Фактически

вся работа выполняется функцией `processRequest()`, поэтому разные случаи обрабатываются этой функцией.

После анализа формы выводятся строковые константы `pageTop` (здесь не показана) и `form`. Если в ходе анализа обнаружится ошибка, также будет выведено сообщение об ошибке; `anError` – это строка формата, а `err` – значение ошибки для форматирования. (О форматировании строк подробно рассказывается в §3.5.)

```
anError = `<p class="error">%s</p>`
```

В случае успешного выполнения анализа (как и должно быть) вызывается функция `processRequest()`, извлекающая числа, введенные пользователем. Если все числа окажутся допустимыми, вызовом функции `getStats()`, которая была показана выше, вычисляются и выводятся в форматированном виде статистические характеристики; иначе будет выведено сообщение об ошибке, если оно имеется. (Когда форма отображается первый раз, в ней нет чисел, и никаких ошибок пока не произошло, в этом случае переменная `ok` хранит значение `false`, а переменная `message` – пустую строку.) И в конце функции выводится строковая константа `pageBottom` (здесь не показана), закрывающая теги `<body>` и `<html>`.

```
func processRequest(request *http.Request) ([]float64, string, bool) {
    var numbers []float64
    if slice, found := request.Form["numbers"]; found && len(slice) > 0 {
        text := strings.Replace(slice[0], ",", " ", -1)
        for _, field := range strings.Fields(text) {
            if x, err := strconv.ParseFloat(field, 64); err != nil {
                return numbers, "" + field + " is invalid", false
            } else {
                numbers = append(numbers, x)
            }
        }
    }
    if len(numbers) == 0 {
        return numbers, "", false // при первом отображении данные отсутствуют
    }
    return numbers, "", true
}
```

Эта функция читает данные формы из значения `request`. Если форма отображается первый раз, элемент `<input>` с именем «numbers»

еще пуст. Это не является ошибкой, поэтому возвращается пустой срез со значениями типа `float64`, пустое сообщение об ошибке и `false`, чтобы показать, что статистические характеристики не были вычислены, — в результате этого будет отображена пустая форма. Если пользователь ввел некоторые данные, возвращается срез со значениями типа `float64`, пустое сообщение об ошибке и `true`; или если одно или более чисел оказались недопустимыми — пустой (возможно) срез, сообщение об ошибке и `false`.

Значение `request` имеет поле `Form` типа `map[string][]string` (§4.3). Это означает, что ключи отображения являются строками, а значения — срезами со строками. То есть одному ключу соответствует значение, содержащее произвольное количество строк. Например, если пользователь введет числа «5 8.2 7 13 6», отображение `Form` будет иметь ключ `"numbers"` со значением `[]string{"5 8.2 7 13 6"}`, то есть его значением будет срез со строками, фактически содержащий единственную строку. (Для сравнения — следующий пример среза содержит две строки: `[]string{"1 2 3", "a b c"}`.) Функция проверяет наличие ключа `"numbers"` (он должен быть) и, если он присутствует и его значение содержит хотя бы одну строку, можно быть уверенными, что в форме имеются числа для чтения.

Строка чисел, введенных пользователем, извлекается с помощью функции `strings.Replace()`, при этом попутно запятые заменяются пробелами. (Третий аргумент определяет максимальное число замен; `-1` означает неограниченное число замен.) После извлечения строки с числами, разделенными пробелами, она разбивается функцией `strings.Fields()` (по любым последовательностям пробельных символов) и возвращается в виде среза, по которому затем выполняются итерации с помощью цикла `for ...range`. (Функции из пакета `strings` рассматриваются в §3.6; цикл `for ...range` рассматривается в §5.3.) Для каждой строки («5», «8.2» и др.) предпринимается попытка преобразовать строку в значение типа `float64` с помощью функции `strconv.ParseFloat()`, принимающей строку и размер результата в битах, 32 или 64 (§3.6). Если в процессе преобразования возникнет ошибка, вызывающей программе немедленно возвращаются все числа типа `float64`, которые удалось получить, непустое сообщение об ошибке и значение `false`. В случае успешного преобразования полученное число типа `float64` добавляется в конец среза `numbers`. Встроенная функция `append()` принимает срез и одно или более значений и возвращает срез, содержащий

все элементы из оригинального среза, плюс значения аргументов. Функция достаточно интеллектуальна, чтобы повторно использовать оригинальный срез, если его емкость больше длины, что обеспечивает ей высокую эффективность. (Подробнее функция `append()` рассматривается в §4.2.3.)

Если функция не завершилась внутри цикла (встретив недопустимое число), она вернет числа, пустое сообщение об ошибке и `true`, в противном случае, если числа отсутствуют (при первом отображении формы), возвращается `false`.

```
func formatStats(stats statistics) string {  
    return fmt.Sprintf(`<table border="1">  
<tr><th colspan="2">Results</th></tr>  
<tr><td>Numbers</td><td>%v</td></tr>  
<tr><td>Count</td><td>%d</td></tr>  
<tr><td>Mean</td><td>%f</td></tr>  
<tr><td>Median</td><td>%f</td></tr>  
</table>`, stats.numbers, len(stats.numbers), stats.mean, stats.median)  
}
```

После вычисления статистических характеристик их нужно вывести, а поскольку программа является веб-приложением, требуется создать разметку HTML. (В стандартной библиотеке Go имеются специализированные пакеты `text/template` и `html/template` для создания текстовых и HTML-шаблонов, управляемых данными, но здесь разметка настолько проста, что проще было создать ее вручную. Небольшой пример, использующий пакет `text/template`, демонстрируется в §9.4.2.)

Функция `fmt.Sprintf()` принимает строку формата и одно или более значений и возвращает строку, являющуюся копией строки формата, в которой спецификаторы (такие как `%v`, `%d`, `%f`) замещены соответствующими значениями. (Подробнее о форматировании строк рассказывается в §3.5.) Здесь не требуется предусматривать экранирование HTML-элементов, поскольку все значения являются числами. (Если бы потребовалось экранирование, можно было бы использовать функцию `template.HTMLEscape()` или `html.EscapeString()`.)

Как демонстрирует этот пример, язык Go позволяет легко разрабатывать веб-приложения за счет создания простой разметки HTML вручную и предоставляет пакеты `html`, `net/http`, `html/template` и `text/template`, еще больше облегчающие задачу.

2.5. Упражнения

В этой главе предлагается решить два упражнения, и оба связаны с числами. В первом упражнении потребуется изменить программу `statistics`, представленную выше; во втором – создать с нуля простое математическое веб-приложение.

1. Скопируйте содержимое каталога `statistics`, например, в каталог `my_statistics` и измените файл `my_statistics/statistics.go`, добавив реализацию вычисления дополнительных статистических характеристик: модального значения и стандартного отклонения. Когда пользователь щелкнет на кнопке **Calculate** (Рассчитать), программа выведет данные, как показано на рис. 2.3.

Для этого придется добавить в структуру `statistics struct` два

дополнительных элемента и реализовать две новые функции, выполняющие вычисления соответствующих характеристик. В решение, которое приводится в файле `statistics_ans/statistics.go`, было добавлено около 40 строк кода и задействована встроенная функция `append()` (§4.2.3) для добавления чисел в срез. Функция, вычисляющая стандартное отклонение, весьма проста в реализации – она использует некоторые функции из пакета `math` и занимает не более десяти строк. Стандартное

отклонение вычисляется по формуле $\sigma = \sqrt{\frac{\sum (x - \bar{x})^2}{n - 1}}$, где x – каждое число в последовательности, \bar{x} – среднее, а n – количество чисел.

Модальное значение – это наиболее часто встречающееся число или числа, если имеются два или более чисел, встречающихся с одинаковой наибольшей частотой. Предложенное решение не возвращает модального значения, если все числа встречаются с одинаковой частотой. Определение модального значения сложнее, чем стандартного отклонения, и его реализация занимает примерно 20 строк.

2. Создайте веб-приложение для решения квадратного уравнения с использованием стандартной формулы $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$.

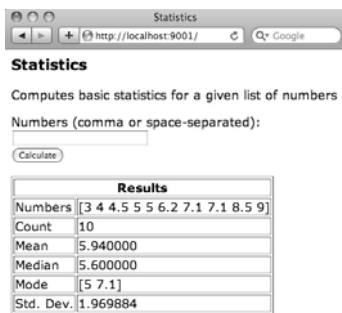


Рис. 2.3. Решение первого упражнения в Mac OS X

Для поиска решения используйте комплексные числа, чтобы было возможно найти решение даже при отрицательном дискриминанте (часть формулы $b^2 - 4ac$). Сначала добейтесь корректной работы математической части программы, чтобы она выводила страницу, как показано рис. 2.4 слева. Затем измените приложение, реализовав в нем более интеллектуальный вывод результатов, как показано на рис. 2.4 справа.

Для начала скопируйте функции `main()`, `homePage()` и `processRequest()` из приложения `statistics`, добавьте в функцию `homePage()` вызов трех новых функций, `formatQuestion()`, `solve()` и `formatSolutions()`, и перепишите функцию `processRequest()` так, чтобы она читала три отдельных вещественных числа. Файл `quadratic_ans1/quadratic.go` содержит начальную реализацию приложения, занимающую примерно 120 строк. Эта версия достаточно интеллектуальна, чтобы вывести только одно решение, когда оба корня уравнения примерно равны, для чего используется функция `EqualFloat()`, обсуждавшаяся выше в этой главе.

Вторая версия приложения, находящаяся в файле `quadratic_ans2/quadratic.go`, занимает примерно 160 строк и предусматривает форматирование результатов в зависимости от их значений. Например, в этой версии предусматривается замена «+ -» на «-» и «1x» на «x», подавляется вывод членов с нулевыми коэффициентами (например, «0x»), а значения корней выводятся как обычные вещественные числа, если мнимые части приблизительно равны нулю. В этой версии используются функции из пакета `math/cmplx`, такие как `cmplx.IsNaN()`, и более сложные операции форматирования строк (§3.5).

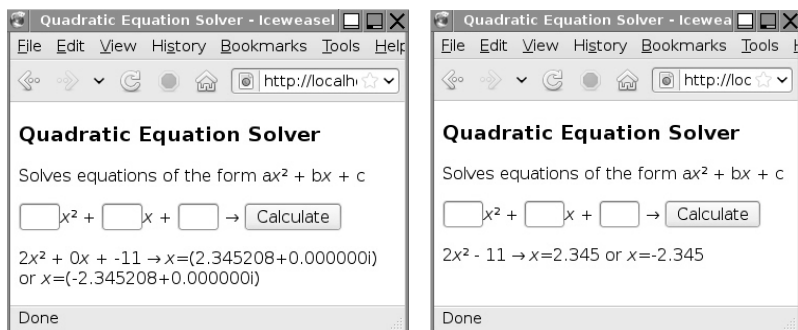


Рис. 2.4. Приложение для решения квадратных уравнений в Linux



3. Строки

В этой главе рассматриваются тип `string` и основные пакеты из стандартной библиотеки для работы со строками. В отдельных разделах главы рассказывается, как записываются строковые литералы и как используются строковые операторы, как индексировать строки и извлекать срезы (подстроки) из строк и как форматировать вывод строк, чисел и других значений, встроенных и пользовательских типов.

Высокоуровневые средства языка Go для работы со строками, такие как цикл `for ...range`, позволяющий выполнять итерации по символам строк, функции из пакетов `strings` и `strconv` и возможность извлечения срезов строк, включают все необходимое для решения повседневных задач. Тем не менее в этой главе подробно рассматриваются особенности строк в языке Go, включая низкоуровневые, такие как внутреннее представление строк. Низкоуровневые аспекты строк представляют определенный интерес, и их знание может пригодиться в некоторых ситуациях.

Юникод

До появления Юникода было невозможно обеспечить сохранение в одном файле текста на разных языках, например включить в текст на английском языке цитаты на японском и русском языках, поскольку для разных языков использовались разные кодировки, и для каждого текстового файла могла использоваться только одна кодировка.

Юникод обеспечивает возможность представления символов из всех известных систем письменности, поэтому один текстовый файл, для которого используется кодировка Юникод, может содержать текст на смеси самых разных языков, а также математические, полиграфические и другие специальные символы.

Каждый символ Юникода имеет уникальный идентификационный номер, называемый *кодovým пунктом* (code point). Всего определено более 100 000 символов Юникода с кодowymi пунктами в диапазоне от `0x0` до `0x10FFFF` (последний определен в языке Go как константа `unicode.MaxRune`), с некоторыми промежуточными и специальными областями. Согласно документации к Юникоду, кодové пункты записываются четырьмя шестнадцатеричными цифрами в форме `U+hhhh`, например: `U+21D4` – для символа ⇨.

Отдельные кодовые пункты (то есть символы) в языке Go представлены значениями типа `rune`. (Тип `rune` является синонимом типа `int32`; §2.3.1.)

Текст Юникода в файлах или в памяти должен быть представлен с применением некоторой кодировки. Стандарт Юникода определяет различные форматы преобразования Юникода (Unicode Transformation Formats), или кодировки, такие как UTF-8, UTF-16 и UTF-32. В языке Go для представления строк используется кодировка UTF-8. UTF-8 – это наиболее широко используемая кодировка – она фактически является стандартом кодирования текстовых файлов и по умолчанию используется для файлов в форматах XML и JSON.

Для представления каждого кодового пункта в кодировке UTF-8 может использоваться от одного до четырех байт. Для строк, содержащих только 7-битные символы ASCII (US-ASCII), байты и символы отображаются друг в друге непосредственно, потому что каждый 7-битный символ ASCII представлен в кодировке UTF-8 единственным байтом (с тем же значением). Как следствие текст на английском языке в кодировке UTF-8 хранится очень компактно (на каждый символ отводится единственный байт); как другое следствие – текстовый файл в 7-битной кодировке ASCII ничем не отличается от текстового файла с тем же текстом в кодировке UTF-8.

Строки в языке Go являются неизменяемыми последовательностями произвольных байтов. В большинстве случаев байты в строках представляют текст Юникода в кодировке UTF-8 (см. врезку «Юникод» выше). Наличие поддержки Юникода означает, что строки в языке Go могут содержать текст на смеси любых языков, известных в мире, без ограничений, накладываемых кодовыми страницами.

Тип `string` в языке Go в корне отличается от эквивалентных типов во многих других языках. Тип `String` в языке Java, `std::string` в языке C++ и `str` – в Python 3 – все они являются последовательностями символов *фиксированного* размера (с некоторыми ограничениями), тогда как строки в языке Go являются последовательностями символов *переменного* размера, где каждый символ может быть представлен одним или более байтами, обычно в кодировке UTF-8.

На первый взгляд может показаться, что строки в других языках удобнее, чем строки в языке Go, потому что они позволяют обращаться к символам непосредственно, что в языке Go возможно, только если строка состоит исключительно из 7-битных символов ASCII (так как в кодировке UTF-8 каждый из них представлен единственным байтом). Однако с практической точки зрения подобное отличие не является проблемой для программистов на языке Go: во-первых, потому что прямое обращение к символам нечасто бывает необходимо

благодаря поддержке итераций по символам строк; во-вторых, потому что стандартная библиотека содержит исчерпывающий набор функций для работы со строками; и в-третьих, потому что в языке Go строки всегда можно преобразовать в срезы с кодовыми пунктами Юникода (типа `[]rune`) и обращаться к ним непосредственно.

Хранение текста в кодировке UTF-8 имеет определенные преимущества, по сравнению, скажем, со строками в языке Java или Python, каждый из которых также поддерживает строки Юникода. В языке Java строки представлены последовательностями кодовых пунктов, каждый из которых занимает 16 бит; в Python, в версиях с 2.x по 3.2, используется тот же подход, но в нем для хранения кодовых пунктов отводятся 16 или 32 бита, в зависимости от параметров сборки Python. Это означает, что для текста на английском языке в Go отводится по 8 бит на каждый символ, тогда как в Java и Python – как минимум в два раза больше. Еще одно преимущество заключается в том, что при использовании кодировки UTF-8 аппаратный *порядок следования байтов* не имеет значения, тогда как при использовании кодировок UTF-16 и UTF-32 его необходимо учитывать (например, чтобы обеспечить корректное декодирование текста, может потребоваться использовать кодировку UTF-16 с обратным порядком следования байт). Кроме того, учитывая, что кодировка UTF-8 фактически стала стандартом кодирования текстовых файлов во всем мире, в других языках программирования приходится кодировать и декодировать такие файлы, чтобы обеспечить преобразование их содержимого во внутреннее представление Юникода и обратно, тогда как в Go можно непосредственно читать и записывать такие файлы. Помимо этого, некоторые крупные библиотеки (такие как GTK+) уже используют строки символов в кодировке UTF-8, поэтому программы на языке Go могут работать с ними, минуя этап кодирования/декодирования.

Фактически строки в языке Go столь же удобны и просты в использовании, как и в других языках. Это станет очевидно, как только вы познакомитесь с идиомами работы со строками в языке Go.

3.1. Литералы, операторы и экранированные последовательности

Строковые литералы определяются с помощью кавычек (") или обратных апострофов ('). Кавычки используются для определения *интерпретируемых строковых литералов* – такие строки поддерживают

экранированные последовательности, перечисленные в табл. 3.1 (ниже), но они не могут занимать несколько строк в программе. Обратные апострофы используются для определения обычных строковых литералов, такие строки могут занимать несколько строк в программе, но они не поддерживают экранированных последовательностей и могут содержать любые символы, кроме обратных апострофов. Интерпретируемые строковые литералы используются чаще, но для записи многострочных сообщений, разметки HTML и регулярных выражений удобнее использовать строковые литералы в обратных апострофах. Ниже приводятся несколько примеров литералов.

```
text1 := "\"what's that?\"", he said" // Интерпретируемый строковый литерал
text2 := "'what's that?', he said' // Простой строковый литерал
radicals := "\u221A \U0000221a" // radicals == "√ √ √"
```

Таблица 3.1. Экранированные последовательности

Экранированная последовательность	Описание
\\	Обратный слеш (\)
\000	Символ Юникода, соответствующий 8-битному кодовому пункту, заданному тремя восьмеричными цифрами
\'	Апостроф ('), может использоваться только внутри символьных литералов
\"	Кавычки ("), может использоваться только внутри интерпретируемых строковых литералов
\a	ASCII-символ «сигнал» (BEL)
\b	ASCII-символ «забой» (BS)
\f	ASCII-символ «перевод формата» (FF)
\n	ASCII-символ «перевод строки» (LF)
\r	ASCII-символ «возврат каретки» (CR)
\t	ASCII-символ «табуляция» (TAB)
\uhhhh	Символ Юникода, соответствующий 16-битному кодовому пункту, заданному четырьмя шестнадцатеричными цифрами
\Uhhhhhhhh	Символ Юникода, соответствующий 32-битному кодовому пункту, заданному восемью шестнадцатеричными цифрами
\v	ASCII-символ «вертикальная табуляция» (VT)
\xhh	Символ Юникода, соответствующий 8-битному кодовому пункту, заданному двумя шестнадцатеричными цифрами

В этом примере были созданы три переменные типа `string`, при этом переменные `text1` и `text2` содержат один и тот же текст. Поскольку для файлов с расширением `.go` используется кодировка UTF-8, в них можно включать любые символы Юникода. Однако сохраняется возможность использовать экранированные последовательности Юникода, как это сделано для второго и третьего символов `√`. Здесь невозможно использовать 8-битное восьмеричное или шестнадцатеричное представление кодового пункта, так как они ограничены диапазоном от `U+0000` до `U+00FF`, слишком узкого для представления кодового пункта `U+221A`, соответствующего символу `√`.

Если потребуется определить длинный и интерпретируемый строковый литерал, разместив его на нескольких строках в тексте программы, можно разбить его на несколько литералов и объединить их оператором конкатенации (`+`). Кроме того, несмотря на то что строки в языке Go являются *неизменяемыми*, они поддерживают оператор добавления `+=`. Он замещает имеющуюся строку результатом конкатенации двух строк, если емкости исходной строки недостаточно для размещения добавляемой строки. Эти операторы перечислены в табл. 3.2 (ниже). Строки могут сравниваться с помощью операторов сравнения (см. табл. 2.3 выше). Ниже демонстрируется использование этих операторов:

```
book := "The Spirit Level" +                               // Конкатенация строк
      " by Richard Wilkinson"
book += " and Kate Pickett"                                // Добавление в конец строки
fmt.Println("Josey" < "José", "Josey" == "José") // Сравнение строк
```

В результате выполнения этого фрагмента переменная `book` будет содержать текст «The Spirit Level by Richard Wilkinson and Kate Pickett», а в поток `os.Stdout` будет выведена строка «true false».

3.2. Сравнение строк

Как уже отмечалось, строки в языке Go поддерживают обычные операторы сравнения (`<`, `<=`, `==`, `!=`, `>`, `>=`), перечисленные в табл. 2.3 (выше). *Сравнение строк* этими операторами выполняется побайтно. Строки могут сравниваться непосредственно, например на равенство, и косвенно, например когда оператор `<` используется для сравнения строк с целью сортировки содержимого среза `[]string`. К сожалению,

Таблица 3.2. Операции со строками

Оператор `[]` извлечения среза без всяких ограничений может применяться только к строкам из 7-битных символов ASCII, во всех остальных случаях необходимо проявлять осторожность (см. §3.4 ниже). Строки могут сравниваться с помощью стандартных операторов сравнения: `<`, `<=`, `==`, `!=`, `>=`, `>` (см. табл. 2.3 выше и §3.2 ниже.)

Операция	Описание/результат
<code>s += t</code>	Добавляет строку <code>t</code> в конец строки <code>s</code>
<code>s + t</code>	Конкатенация строк <code>s</code> и <code>t</code>
<code>s[n]</code>	Байт (значение типа <code>uint8</code>) в позиции <code>n</code> , в строке <code>s</code>
<code>s[n:m]</code>	Подстрока, извлеченная из строки <code>s</code> , начиная с позиции <code>n</code> по <code>m-1</code>
<code>s[n:]</code>	Подстрока, извлеченная из строки <code>s</code> , начиная с позиции <code>n</code> по <code>len(s)-1</code>
<code>s[:m]</code>	Подстрока, извлеченная из строки <code>s</code> , начиная с позиции <code>0</code> по <code>m-1</code>
<code>len(s)</code>	Количество байт в строке <code>s</code>
<code>len([] rune(s))</code>	Количество символов в строке <code>s</code> ; то же значение можно получить намного быстрее с помощью функции <code>utf8.RuneCountInString()</code> ; см. табл. 3.10 (ниже)
<code>[]rune(s)</code>	Преобразует строку в срез кодовых пунктов Юникода
<code>string(символы)</code>	Преобразует значение типа <code>[]rune</code> или <code>[]int32</code> в строку; предполагается, что значения типа <code>rune</code> или <code>int32</code> представляют кодовые пункты Юникода ¹ .
<code>[]byte(s)</code>	Преобразует строку <code>s</code> типа <code>string</code> в срез байт без копирования; нет никакой гарантии, что байты будут допустимыми значениями в кодировке UTF-8
<code>string(байты)</code>	Преобразует значение типа <code>[]byte</code> или <code>[]uint8</code> в строку типа <code>string</code> без копирования; не требуется, чтобы байты были допустимыми значениями в кодировке UTF-8
<code>string(i)</code>	Преобразует значение <code>i</code> любого целочисленного типа в значение типа <code>string</code> ; предполагается, что <code>i</code> представляет кодовый пункт Юникода; например, если <code>i == 65</code> , программе будет возвращено значение «А»
<code>strconv.Itoa(i)</code>	Вернет строковое представление числа <code>i</code> типа <code>int</code> и значение ошибки; например если <code>i == 65</code> , программе будут возвращены два значения («65», <code>nil</code>); см. также табл. 3.8 и 3.9 (ниже)
<code>fmt.Sprintf(x)</code>	Вернет строковое представление значения <code>x</code> любого типа; например если <code>x</code> является целым числом, равным 65, программе будет возвращена строка «65»; см. также табл. 3.3 (ниже)

при выполнении сравнения могут возникать три проблемы. Эти проблемы проявляются во всех языках программирования,

¹ Преобразование всегда выполняется успешно; недопустимые целочисленные значения преобразуются в символ замены Юникода с кодовым пунктом U+FFFD, который часто изображается как «?»

поддерживающих строки Юникода, и не являются характерными только для языка Go.

Первая проблема – в том, что некоторые символы Юникода могут быть представлены двумя и более разными последовательностями байт. Например, символ \AA может обозначать единицу измерения расстояний «ангстрем» или быть простым символом \AA с кружочком над ним – оба символа часто не различимы на глаз. Символу «ангстрем» в Юникоде соответствует кодовый пункт U+212B, а символу \AA с кружочком над ним – кодовый пункт U+00C5 или два кодовых пункта U+0041 (\AA) и U+030A ($^{\circ}$ – дополнительный кружок сверху). В кодировке UTF-8 символ «ангстрем» (\AA) представляет последовательность байтов [0xE2, 0x84, 0xAB], символ \AA – последовательность [0xC3, 0x85], а символ \AA с дополнительным символом $^{\circ}$ – последовательность [0x41, 0xCC, 0x81]. Разумеется, с точки зрения пользователя, оба символа \AA ничем не отличаются и при сравнении должны определяться как одинаковые, независимо от того, какими последовательностями байтов они представлены.

Первая проблема не столь существенна, как может показаться, потому что для всех последовательностей байт (то есть строк) в кодировке UTF-8 в языке Go воспроизводятся одни и те же кодовые пункты. Это означает, например, что символ \acute{e} в языке Go, в символьных или в строковых литералах, всегда будет представлен одной и той же последовательностью байтов. И конечно же при работе с текстом, состоящим исключительно из символов ASCII (то есть на английском языке), эта проблема вообще никак не проявляется. И даже при работе с текстом, содержащим не-ASCII символы, проблема возникает, только когда существуют два разных символа, имеющих одинаковое начертание, или когда байты UTF-8 поступают в программу из внешних источников, использующих допустимые, но другие отображения кодовых пунктов в последовательности байтов. Если это обстоятельство превращается в действительно серьезную проблему, всегда можно написать собственную функцию *нормализации*, гарантирующую, например, что символ \acute{e} всегда будет представлен последовательностью байтов [0xC3, 0xA9] (используемой в языке Go по умолчанию), а не, к примеру, последовательностью [0x65, 0xCC, 0x81] (то есть комбинацией символов e и $^{\circ}$). Принципы нормализации символов Юникода подробно разъясняются в документе «Unicode Normalization Forms» (формы нормализации Юникода) (unicode.org/reports/tr15). На момент написания этих строк в состав стандартной библиотеки языка Go входил *экспериментальный* пакет, реализующий нормализацию (`exp/norm`).

Поскольку первая проблема в действительности проявляется, только когда дело доходит до обработки строк, поступающих из внешних источников, и только если используется иной принцип отображения кодовых пунктов в последовательности байтов, отличный от принятого в языке Go, лучшим ее решением будет изоляция программного кода, принимающего внешние данные. При таком подходе изолированный код мог бы выполнять нормализацию принимаемых строк до того, как они попадут в программу.

Вторая проблема в том, что бывают ситуации, когда пользователи могут вполне обоснованно ожидать, что *разные* символы должны определяться как равные. Например, пользуясь программой, осуществляющей *поиск по тексту*, пользователь может ввести слово «file». Естественно, он надеется, что программа отыщет все вхождения слова «file», а также все вхождения «fi-le» (то есть слог «fi» в конце одной строки, за которым следует слог «le» в начале следующей). Аналогично он может надеяться отыскать по строке поиска «5» все вхождения «5», «₅», «⁵» и даже «Ⓢ». Как и первая, эта проблема решается с помощью нормализации некоторого вида.

Третья проблема состоит в том, что *порядок сортировки* некоторых символов зависит от языка, на котором набран текст. Например, в шведском алфавите символ *ä* следует за символом *z*, тогда как в телефонных справочниках на немецком языке символ *ä* при сортировке соответствует паре символов *ae*, а в словарях немецкого языка он соответствует символу *a*. Еще один пример: в английском языке символ *ø* при сортировке соответствует символу *o*, а в датском и норвежском языках он должен следовать за символом *z*. Существует не только огромное количество правил, подобных упомянутым, но они еще осложняются тем обстоятельством, что иногда одно и то же приложение могут использовать люди разных национальностей (ожидающие получить разный порядок сортировки). Кроме того, иногда строки могут содержать слова на разных языках (например, несколько слов на испанском и несколько слов на английском), а к некоторым символам (таким как стрелки, типографские знаки и математические символы) вообще неприменимо понятие порядка следования.

Но большим плюсом является то, что в языке Go строки сравниваются побайтно, в соответствии с порядком сортировки символов ASCII. И при сравнении строк, состоящих только из символов нижнего или верхнего регистра, будет получаться порядок сортировки, более естественный для английского языка, как будет показано в примере ниже (§4.2.4).

3.3. Символы и строки

Символы в языке Go могут быть представлены двумя разными (но взаимозаменяемыми) способами. Единственный символ может быть представлен значением типа `rune` (или `int32`). С этого момента термины «символ», «кодový пункт», «символ Юникода» и «кодový пункт Юникода» будут использоваться взаимозаменяемо для ссылки на значение типа `rune` (или `int32`), хранящее единственный символ. Строки в языке Go представлены последовательностями из нуля или более символов – каждый символ внутри строки представлен одним или более байт в кодировке UTF-8.

С помощью операции преобразования типа (`string(символ)`) единственный символ можно преобразовать в односимвольную строку. Например:

```
æs := ""
for _, char := range []rune{'æ', 0xE6, 0346, 230, '\xE6', '\u00E6'} {
    fmt.Printf("[0x%X '%c'] ", char, char)
    æs += string(char)
}
```

Этот фрагмент выведет строку, в которой текст `[0xE6 'æ']` повторяется шесть раз, а после его выполнения переменная `æs` будет содержать строку, содержащую текст `ææææææ`. (Более эффективные альтернативы оператору `+=` для использования в цикле будут показаны чуть ниже.)

Преобразовать строку в срез со значениями типа `rune` (то есть кодовых пунктов) можно с помощью операции преобразования `chars := []rune(s)`, где `s` – значение типа `string`. Значение `chars` в этом случае будет иметь тип `[]int32`, поскольку тип `rune` является синонимом типа `int32`. Такая возможность может пригодиться, например, когда потребуется выполнить посимвольный анализ строки и при этом выбирать символы, стоящие перед и после текущего. Обратное преобразование выполняется так же просто: `s := string(chars)`, где значение `chars` имеет тип `[]rune`, или `[]int32`, а значение `s` будет иметь тип `string`. Оба преобразования имеют определенные накладные расходы, но выполняются достаточно быстро (имеют сложность $O(n)$, где n – количество байт; см. врезку «Нотация $O(\dots)$ » ниже). Дополнительные операции преобразования строк перечислены в табл. 3.2 (выше), а преобразования «число↔строка» – в табл. 3.8 и табл. 3.9 (ниже).

Несмотря на удобство, оператор `+=` обеспечивает не самый эффективный способ наращивания строк в циклах. Более удачный способ (хорошо знакомый программистам на Python) заключается в заполнении среза со строками (`[]string`) с последующим объединением его элементов вызовом функции `strings.Join()`. Однако в языке Go существует еще более *эффективный путь*, напоминающий использование класса *StringBuilder* в языке Java. Например:

```
var buffer bytes.Buffer
for {
    if piece, ok := getNextValidString(); ok {
        buffer.WriteString(piece)
    } else {
        break
    }
}
fmt.Print(buffer.String(), "\n")
```

Фрагмент начинается с создания пустого значения типа `bytes.Buffer`. Затем выполняется запись каждой строки в буфер с помощью его метода `bytes.Buffer.WriteString()`. (При необходимости можно было бы также организовать запись строки-разделителя между фрагментами.) В конце вызывается метод `bytes.Buffer.String()`, извлекающий окончательную строку. (Особенности использования мощного и гибкого типа `bytes.Buffer` будут показаны ниже.)

Прием накопления строки в `bytes.Buffer` потенциально более эффективен с точки зрения вычислительных ресурсов и потребляемой памяти, чем прием на основе оператора `+=`, особенно при большом количестве объединяемых строк.

Цикл `for ...range` (§5.3) с успехом можно использовать для итераций по символам строки. В этом случае в каждой итерации программе становятся доступны индекс текущей позиции в строке и кодовый пункт в этой позиции. Ниже приводятся пример использования этой версии цикла и вывод, полученный в результате выполнения данного фрагмента.

phrase := "vått og tørt"	string: "vått og tørt"
fmt.Printf("string: \"%s\\n", phrase)	index rune char bytes
fmt.Println("index rune char bytes")	0 U+0076 'v' 76
for index, char := range phrase {	1 U+00E5 'å' C3 A5
fmt.Printf("%-2d %U '%c' % X\\n",	3 U+0074 't' 74
index, char, char,	4 U+0074 't' 74

}	[byte(string(char)))	5	U+0020	' '	20
		6	U+006F	'o'	6F
		7	U+0067	'g'	67
		8	U+0020	' '	20
		9	U+0074	't'	74
		10	U+00F8	'ø'	C3 B8
		12	U+0072	'r'	72
		13	U+0074	't'	74

Нотация $O(\dots)$

Нотация $O(\dots)$ используется в теории сложности алгоритмов для описания эффективности и потребления памяти конкретными алгоритмами. В большинстве случаев в скобках указываются значения в пропорциях к n – числу обрабатываемых элементов или длине обрабатываемого элемента. В скобках также может указываться мера потребления памяти или время обработки.

Запись $O(1)$ означает постоянное время, то есть время обработки не зависит от величины n . Запись $O(\log n)$ означает увеличение времени по логарифмическому закону – это очень быстрый алгоритм, время работы которого пропорционально $\log n$. Запись $O(n)$ означает линейное увеличение времени – это довольно быстрый алгоритм, время работы которого пропорционально n . Запись $O(n^2)$ означает увеличение времени по квадратичному закону – это медленный алгоритм, время работы которого пропорционально n^2 . Запись $O(n^m)$ означает увеличение времени по полиномиальному закону – скорость работы такого алгоритма падает очень быстро с ростом n , особенно при значениях $m \geq 3$. Запись $O(n!)$ означает увеличение времени по факториальному закону – даже при маленьких значениях n такой алгоритм становится слишком медленным, чтобы иметь практическую ценность.

В этой книге обозначение $O(\dots)$ используется в разных местах, чтобы дать некоторое представление о стоимости тех или иных операций, например преобразования значения типа `string` в значение типа `[] rune`.

В начале фрагмента создается строковый литерал `phrase` и в следующей строке выводится на экран. Затем выполняются итерации по *символам* в строке – в языке Go цикл `for ... range` автоматически декодирует байты UTF-8 в кодовые пункты Юникода (значения типа `rune`), поэтому нет необходимости беспокоиться о внутреннем их представлении. Для каждого символа выводится номер его позиции, значение кодового пункта (в форме записи, принятой в стандарте Юникода), сам символ и соответствующие ему байты в кодировке UTF-8.

Чтобы получить список байтов, кодовые пункты (значения `char` типа `rune`) преобразуются в строку (содержащую единственный символ, который состоит из одного или более байтов в кодировке UTF-8). Затем эта односимвольная строка преобразуется в значение типа `[]byte`, то есть в срез с байтами, благодаря чему появляется возможность доступа к фактическим байтам. Преобразование `[]byte(string)` выполняется очень быстро ($O(1)$), так как `[]byte` просто ссылается на внутреннее представление строки `string`, без необходимости копировать какие-либо данные. То же справедливо и для обратного преобразования `string([]byte)` – здесь байты внутреннего представления строки также никуда не копируются, поэтому данное преобразование тоже имеет сложность $O(1)$. Преобразования между строками и последовательностями байтов перечислены в табл. 3.2 (выше).

Спецификаторы формата `%-2d`, `%U`, `%c` и `%X` описываются ниже (§3.5). Как будет показано далее, спецификатор `%X` используется для вывода целых чисел в шестнадцатеричном виде, а когда он применяется к значению `[]byte`, выводится последовательность чисел, состоящих из двух шестнадцатеричных цифр, по одному на каждый байт. Наличие пробела в спецификаторе указывает, что байты должны выводиться через пробел.

На практике циклы `for ...range`, используемые для итераций по символам строк, наряду с функциями из пакетов `strings` и `fmt` (и в меньшей степени из пакетов `strconv`, `unicode` и `unicode/utf8`) обеспечивают все, что необходимо для обработки строк. Однако, помимо этого, тип `string` поддерживает возможность создания срезов (поскольку во внутреннем представлении строка фактически является значением типа `[]byte`), что может оказаться весьма полезным, так как исключает вероятность разрыва многобайтных символов пополам при обработке!

3.4. Индексирование и получение срезов строк

Как видно из табл. 3.2 (выше), язык Go поддерживает операцию получения срезов строк, используя синтаксис, напоминающий синтаксис языка Python. Этот синтаксис можно использовать для получения срезов значений любых типов, как будет показано в главе 4.

Поскольку строки в языке Go хранят текст в виде байтов в кодировке UTF-8, необходимо соблюдать меры предосторожности, чтобы при создании срезов не нарушить границ символов. В этом

нет ничего сложного при работе с текстом, состоящим из 7-битных символов ASCII, поскольку каждый символ представлен единственным байтом, но в других случаях ситуация может оказаться намного более сложной, так как символы могут быть представлены одним и более байтами. Как правило, в обычной практике вообще не требуется извлекать срезы строк – достаточно иметь простую возможность итераций по символам в цикле `for ...range`, но иногда действительно бывает необходимо получить срез, чтобы извлечь подстроку. Один из способов, гарантирующих целостность границ символов при извлечении среза, заключается в использовании функций из пакета `strings`, таких как `strings.Index()` или `strings.LastIndex()`. Функции, входящие в пакет `strings`, перечислены в табл. 3.6 и табл. 3.7 (ниже).

Для начала рассмотрим различные способы представления строки. Отсчет индексов, то есть позиций байтов UTF-8 в строке, начинается с 0 и продолжается до значения, определяющего длину строки минус единицу. Также имеется возможность индексирования в обратном направлении – с конца строки, с использованием индексов со значениями `len(s) - n`, где `n` – количество байтов, отсчитываемых с конца. Например, для выражения `s := "naïve"`, на рис. 3.1 показана строка `s` в виде последовательностей символов Юникода, кодовых пунктов и байтов, а также приводятся несколько допустимых индексов и пара срезов.

Для доступа к каждой позиции в строке, изображенной на рис. 3.1, можно использовать оператор индексирования `[]`, который возвратит соответствующий ASCII-символ (как значение типа `byte`). Например, `s[0] == 'n'`, а `s[len(s) - 1] == 'e'`. *Первый* байт последовательности, соответствующей символу *ï*, имеет индекс 2, но, если обратиться к элементу строки `s[2]`, программа получит только первый байт (0xC3) символа *ï* в кодировке UTF-8 – подобное редко требуется в программах.

s[:2]		s[2:] == s[len(s)-4:]				Срезы
'n'	'a'	'ï'		'v'	'e'	Символы
U+006E	U+0061	U+00EF		U+0076	U+0065	Кодовые пункты
0x6E	0x61	0xC3	0xAF	0x76	0x65	Байты
0	1	2	3	4	5	Индексы
		len(s)-2		len(s)-1		

Рис. 3.1. Строение строки

Для строк, содержащих только 7-битные ASCII-символы, первый символ (в виде значения типа `byte`) можно извлечь с помощью выражения `s[0]`, а последний – с помощью выражения `s[len(s) - 1]`. Однако в общем случае для извлечения первого символа (в виде значения типа `rune`, содержащего все байты UTF-8, представляющие символ) следует использовать функцию `utf8.DecodeRuneInString()`, а для извлечения последнего символа – функцию `utf8.DecodeLastRuneInString()` (см. табл. 3.10 ниже).

Для доступа к отдельным символам имеется несколько возможностей. Для строк, содержащих только 7-битные ASCII-символы, можно использовать обычный оператор индексирования `[]`, обеспечивающий очень быстрый ($O(1)$) доступ. В случае с другими строками можно преобразовать строку в значение типа `[]rune` и использовать оператор индексирования `[]` с этим значением. В этом случае индексирование тоже выполняется очень быстро ($O(1)$), но сама операция преобразования является достаточно дорогостоящей, с точки зрения производительности и потребления памяти ($O(n)$).

В случае с примером, представленным выше, если записать инструкцию `chars := []rune(s)`, будет создана переменная `chars`, хранящая срез значений типа `rune` (то есть `int32`) с пятью кодовыми пунктами, представляющими шесть байт, как показано на рис. 3.1. Напомню, что любое значение типа `rune` (кодový пункт) легко можно преобразовать обратно в строку, содержащую единственный символ, с помощью выражения преобразования `string(char)`.

Для произвольных строк (то есть для строк, которые могут содержать неASCII-символы), обычная операция индексирования далеко не всегда дает желаемый результат. Вместо нее следует использовать операцию получения среза строки, который также позволяет получить строку вместо байтов. Для большей надежности извлечения срезов из произвольных строк, определения позиции символа, начиная с которого или заканчивая которым требуется получить срез, лучше использовать функции из пакета `strings` – см. табл. 3.6 и табл. 3.7 (ниже).

Следующее равенство справедливо не только для срезов строк, но и для срезов любых других типов:

```
s == s[:i] + s[i:] // s – это строка; i – значение типа int; 0 <= i <= len(s)
```



Теперь рассмотрим пример извлечения среза с использованием упрощенного подхода. Допустим, что имеется строка, из которой требуется извлечь первое и последнее слово. Ниже представлен простейший способ решения этой задачи:

```

line := "røde og gule sløjfer"
i := strings.Index(line, " ") // Получить индекс первого пробела
firstWord := line[:i] // Получить срез до первого пробела
j := strings.LastIndex(line, " ") // Получить индекс последнего пробела
lastWord := line[j+1:] // Получить срез от последнего пробела
fmt.Println(firstWord, lastWord) // Выведет: røde sløjfer

```

Переменной `firstWord` (типа `string`) будут присвоены байты из строки `line`, начиная с позиции 0 (первый байт) до позиции с индексом `i - 1` (то есть до последнего байта перед пробелом включительно), потому что срезы извлекаются до конечной, указанной позиции, не включая ее. Аналогично переменной `lastWord` будут присвоены байты из строки `line`, начиная с позиции `j + 1` (первый байт после пробела), до последнего байта в строке `line` включительно (то есть до позиции с индексом `len(line) - 1`).

Этот способ прекрасно подходит для случая с пробелами и другими 7-битными ASCII-символами, но он не пригоден для случаев, когда слова отделяются произвольными *пробельными символами* Юникода, такими как U+2028 (Line Separator,  – разделитель строк) или U+2029 (Paragraph Separator,  – разделитель абзацев).

Ниже показан пример поиска первого и последнего слова в строках, где слова могут разделяться произвольными пробельными символами.

```

line := "rå tørt\u2028vær"
i := strings.IndexFunc(line, unicode.IsSpace) // i == 3
firstWord := line[:i]
j := strings.LastIndexFunc(line, unicode.IsSpace) // j == 9
_, size := utf8.DecodeRuneInString(line[j:]) // size == 3
lastWord := line[j+size:] // j + size == 12
fmt.Println(firstWord, lastWord) // Выведет: rå vær

```

Содержимое строки `line` в виде последовательности символов, кодовых пунктов и байтов показано на рис. 3.2. Здесь также показаны номера позиций байтов и срезы, получаемые во фрагменте кода выше.

Функция `strings.IndexFunc()` возвращает индекс первой позиции в строке, определяемой первым аргументом, для которой функция, определяемая вторым аргументом (имеющая сигнатуру `func(rune) bool`), вернет `true`. Функция `strings.LastIndexFunc()` действует аналогично, за исключением того, что она начинает просмотр строки с конца и возвращает индекс последней позиции, для которой функция во втором аргументе вернет `true`. Здесь во втором аргументе

line[:i]			line[j:]										line[j+size:]			Срезы
'r'	'ä'	' '	't'	'ø'	'r'	't'	'раздел. строка'					'v'	'æ'	'r'	Символы	
U+0072	U+00E5	U+0020	U+0074	U+00F8	U+0072	U+0074	U+2028					U+0076	U+00E6	U+0072	Кодовые пункты	
0x72	0xC3 0x45	0x20	0x74	0xC3 0xB8	0x72	0x74	0xE2 0x80 0xA8	0x76	0xC3 0xA6	0x72	Байты					
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	Индексы

Рис. 3.2. Строение строки с пробельными символами

передается функция `IsSpace()` из пакета `unicode` — она принимает кодовый пункт Юникода (типа `rune`) в виде единственного аргумента и возвращает `true`, если он соответствует пробельному символу (см. табл. 3.11 ниже). Имена функций интерпретируются как ссылки на функции, поэтому они могут передаваться в виде параметров другим функциям, при условии что сигнатуры передаваемых функций соответствуют типам параметров (§4.1).

Операции поиска первого пробельного символа с помощью функции `strings.IndexFunc()` и извлечения среза от начала строки до этого символа (не включая его), чтобы получить первое слово, реализуются просто. Но при поиске последнего пробельного символа необходимо быть внимательными, потому что некоторые пробельные символы в кодировке UTF-8 кодируются более чем одним байтом. В данном примере эта проблема решена за счет использования функции `utf8.DecodeRuneInString()`, возвращающей количество байт в первом символе среза строки, начинающегося с последнего пробельного символа. Затем это число добавляется к индексу последнего пробельного символа, чтобы перешагнуть через него, то есть через байты, представляющие этот пробельный символ, и извлекается срез, содержащий только последнее слово.

3.5. Форматирование строк с помощью пакета `fmt`

Пакет `fmt` из стандартной библиотеки языка Go предоставляет семейство функций для вывода данных в виде строк в консоль, в

файлы и в другие значения, реализующие интерфейс `io.Writer`, а также в другие строки. Эти функции перечислены в табл. 3.3 (ниже). Некоторые из функций вывода возвращают значение ошибки. При выводе в консоль значение ошибки обычно игнорируется в программах, но его обязательно следует проверять при выводе данных в файлы, в сетевые соединения и в другие значения¹.

В пакете `fmt` также имеются различные функции ввода (такие как `fmt.Scan()`, `fmt.Scanf()` и `fmt.Scanln()`), предназначенные для чтения данных из консоли, из файлов и из строк. Некоторые из этих функций используются в главе 8 (§8.1.3.2) – см. также табл. 8.2 (ниже). Альтернативой функциям ввода является функция `strings.Fields()`, разбивающая исходную строку на отдельные поля, которые затем можно преобразовать в значения (например, в числа) с помощью функций из пакета `strconv` – см. табл. 3.8 и табл. 3.9 (ниже). Напомню, что в главе 1 мы реализовали чтение данных, вводимых с клавиатуры, за счет создания `bufio.Reader` для чтения из потока `os.Stdin` и использования функции `bufio.Reader.ReadString()`, для извлечения каждой введенной строки (§1.7).

Простейший способ вывода значений заключается в использовании функций `fmt.Print()` и `fmt.Println()` (для вывода в поток `os.Stdout`, то есть в консоль), или функций `fmt.Fprint()` и `fmt.Fprintf()` для вывода в указанное значение типа `io.Writer` (то есть в файл), или функций `fmt.Sprint()` и `fmt.Sprintln()` для вывода в строку.

```
type polar struct{ radius, Θ float64 }
p := polar{8.32, .49}
fmt.Print(-18.5, 17, "Elephant", -8+.7i, 0x3C7, '\u03C7', "a", "b", p)
fmt.Println()
fmt.Println(-18.5, 17, "Elephant", -8+.7i, 0x3C7, '\u03C7', "a", "b", p)
-18.5·17Elephant(-8+0.7i)·967·967ab{8.32·0.49}
-18.5·17·Elephant·(-8+0.7i)·967·967·a·b·{8.32·0.49}
```

Ради ясности, которая особенно необходима при выводе множества следующих друг за другом значений, в листинге вывода каждый пробел был замещен символом `(·)`.

¹ В языке Go также имеются две встроенные функции вывода, `print()` и `println()`. Однако пользоваться ими не рекомендуется, так как они существуют исключительно для удобства разработчиков компилятора языка Go и могут быть удалены в будущем.

Таблица 3.3. Функции вывода в пакете *fmt*

Функция	Описание/результат
<code>fmt.Errorf(format, args...)</code>	Возвращает значение ошибки, содержащее строку, созданную на основе указанной строки формата <code>format</code> и аргументов <code>args</code>
<code>fmt.Fprint(writer, args...)</code>	Выводит аргументы <code>args</code> в поток, определяемый аргументом <code>writer</code> , разделяя нестроковые значения пробелами и используя для каждого спецификатор формата <code>%v</code> ; возвращает число записанных байтов и значение ошибки или <code>nil</code>
<code>fmt.Fprintf(writer, format, args...)</code>	Выводит аргументы <code>args</code> в поток, определяемый аргументом <code>writer</code> , используя указанную строку формата <code>format</code> ; возвращает число записанных байтов и значение ошибки или <code>nil</code>
<code>fmt.Fprintln(writer, args...)</code>	Выводит через пробел аргументы <code>args</code> в поток, определяемый аргументом <code>writer</code> , используя для каждого спецификатор формата <code>%v</code> , разделяя их пробелами и завершая вывод символом перевода строки; возвращает число записанных байтов и значение ошибки или <code>nil</code>
<code>fmt.Print(args...)</code>	Выводит аргументы <code>args</code> в поток <code>os.Stdout</code> , разделяя нестроковые значения пробелами, используя для каждого спецификатор формата <code>%v</code> ; возвращает число записанных байтов и значение ошибки или <code>nil</code>
<code>fmt.Printf(format, args...)</code>	Выводит аргументы <code>args</code> в поток <code>os.Stdout</code> , используя указанную строку формата <code>format</code> ; возвращает число записанных байтов и значение ошибки или <code>nil</code>
<code>fmt.Println(args...)</code>	Выводит через пробел аргументы <code>args</code> в поток <code>os.Stdout</code> , используя для каждого спецификатор формата <code>%v</code> и завершая вывод символом перевода строки; возвращает число записанных байтов и значение ошибки или <code>nil</code>
<code>fmt.Sprint(args...)</code>	Возвращает строку из аргументов <code>args</code> , используя для каждого спецификатор формата <code>%v</code> и разделяя нестроковые значения пробелами
<code>fmt.Sprintf(format, args...)</code>	Возвращает строку из аргументов <code>args</code> , используя указанную строку формата <code>format</code>
<code>fmt.Sprintln(args...)</code>	Возвращает строку из аргументов <code>args</code> , разделенных пробелами, используя для каждого спецификатор формата <code>%v</code> и завершая строку символом перевода строки

Таблица 3.4. Спецификаторы формата для функций вывода из пакета `fmt`

Спецификаторы формата обычно используются для вывода отдельных значений. Если значение является срезом, выводится последовательность его элементов через пробелы, заключенная в квадратные скобки, где каждое значение отформатировано в соответствии с указанным спецификатором. В случае, когда значение является отображением, допускается использовать только спецификатор `%v` или `%#v`, если только ключ и значение не относятся к одному типу, – в этом случае можно также использовать совместимые спецификаторы.

Спецификатор	Описание/результат
<code>%%</code>	Литерал символа <code>%</code>
<code>%b</code>	Целочисленное значение в двоичном представлении (в системе счисления по основанию 2) или (ДОПОЛНИТЕЛЬНО) вещественное число в научной форме записи, где экспонента представлена степенью двойки
<code>%C</code>	Целочисленное значение кодового пункта в виде символа Юникода
<code>%d</code>	Целочисленное значение в десятичном представлении (в системе счисления по основанию 10)
<code>%e</code>	Вещественное или комплексное значение в научной форме записи, где экспонента начинается с символа <code>e</code>
<code>%E</code>	Вещественное или комплексное значение в научной форме записи, где экспонента начинается с символа <code>E</code>
<code>%f</code>	Вещественное или комплексное значение в стандартной форме записи
<code>%g</code>	Вещественное или комплексное значение в формате <code>%e</code> или <code>%f</code> , обеспечивающем наиболее компактное представление
<code>%G</code>	Вещественное или комплексное значение в формате <code>%E</code> или <code>%f</code> , обеспечивающем наиболее компактное представление
<code>%o</code>	Целочисленное значение в восьмеричном представлении (в системе счисления по основанию 8)
<code>%p</code>	Значение адреса в шестнадцатеричном представлении (в системе счисления по основанию 16)
<code>%q</code>	Значение типа <code>string</code> или <code>[]byte</code> в виде строки в кавычках или целочисленное значение в виде строки в апострофах с применением синтаксических правил оформления, принятых в языке Go, если это необходимо
<code>%s</code>	Значение типа <code>string</code> или <code>[]byte</code> в виде последовательности байт кодировки UTF-8; воспроизведет корректный текст Юникода при выводе в текстовый файл или в консоль с поддержкой кодировки UTF-8
<code>%t</code>	Логическое значение как <code>true</code> или <code>false</code>
<code>%T</code>	Тип значения с соблюдением синтаксиса языка Go

Таблица 3.4. Спецификаторы формата для функций вывода из пакета *fmt*

Спецификатор	Описание/результат
%U	Целочисленное значение кодового пункта в форме записи, определяемой стандартом Юникода, по умолчанию включающей четыре цифры, например <code>fmt.Printf("%U", '¶')</code> выведет <code>U+00B60</code>
%v	Значение встроенного или пользовательского типа с использованием формата по умолчанию или значение пользовательского типа с использованием метода <code>String()</code> этого типа, если таковой имеется
%x	Целочисленное значение в виде шестнадцатеричного числа (в системе счисления по основанию 16), а значение типа <code>string</code> или <code>[]byte</code> – в виде последовательности шестнадцатеричных цифр (по две на каждый байт), с использованием символов нижнего регистра <code>a-f</code>
%X	Целочисленное значение в виде шестнадцатеричного числа (в системе счисления по основанию 16), а значение типа <code>string</code> или <code>[]byte</code> – в виде последовательности шестнадцатеричных цифр (по две на каждый байт), с использованием символов верхнего регистра <code>A-F</code>

Таблица 3.5. Модификаторы спецификаторов формата для функций вывода из пакета *fmt*

Модификатор	Описание/результат
пробел	Обеспечивает вывод знака «-» перед отрицательными числами и пробел – перед положительными, добавляет пробел между байтами при использовании спецификатора <code>%x</code> или <code>%X</code> ; например <code>fmt.Printf("% X", "-")</code> выведет <code>E2 86 92</code>
#	Включает «альтернативный» формат вывода: <code>%#o</code> выведет значение в восьмеричном представлении с ведущим 0; <code>%#p</code> выведет значение указателя без ведущих символов 0x; <code>%#q</code> выведет значение типа <code>string</code> или <code>[]byte</code> в виде неинтерпретируемой строки (в обратных апострофах), если возможно, иначе выведет строку в кавычках; <code>%#v</code> выведет значение как есть с использованием синтаксиса языка Go; <code>%#x</code> выведет шестнадцатеричное число с ведущей последовательностью 0x; <code>%#X</code> выведет шестнадцатеричное число с ведущей последовательностью 0X
+	Обеспечивает принудительный вывод знака числа, + или -
-	Обеспечивает выравнивание значения по левому краю (по умолчанию выполняется выравнивание по правому краю)
0	Обеспечивает дополнение слева нулями вместо пробелов

Таблица 3.5. Модификаторы спецификаторов формата для функций вывода из пакета `fmt`

Модификатор	Описание/результат
<code>n.m</code> <code>n</code> <code>.m</code>	<p>Для вещественных или комплексных чисел параметр <code>n</code> (типа <code>int</code>) определяет минимальную ширину поля вывода (может быть увеличено, если использование указанной ширины приведет к усечению числа), а параметр <code>m</code> (типа <code>int</code>) – количество знаков после десятичной точки.</p> <p>Для строк параметр <code>n</code> определяет минимальную ширину поля вывода (если строка окажется короче, перед ней будет добавлено необходимое количество пробелов), а параметр <code>m</code> – максимальное число символов в строке (слева направо), подлежащих выводу, – если строка окажется длиннее, она будет усечена.</p> <p>Любой из параметров, <code>n</code> или <code>m</code>, или оба сразу можно заменить символами «*», в этом случае их значения будут браться из аргументов.</p> <p>Любой из параметров, <code>n</code> или <code>m</code>, можно опустить</p>

Внутри эти функции используют спецификатор формата `%v` (обобщенное значение) и способны выводить значения любых встроенных или пользовательских типов. Например, функции вывода ничего не знают о пользовательском типе `polar`, но благополучно выводят значения это типа.

В главе 6 будет показано, как добавлять метод `String()` в пользовательские типы. Этот метод позволяет организовать вывод значений в каком угодно виде. Если потребуется получить такой же полный контроль над выводом значений встроенных типов, можно воспользоваться функциями вывода, принимающими строку формата в первом аргументе.

Строка формата, которую можно передать функциям `fmt.Errorf()`, `fmt.Printf()`, `fmt.Fprintf()` и `fmt.Sprintf()`, включает один или более *спецификаторов формата*, имеющих вид `%ML`, где `M` определяет один или более модификаторов, а `L` – символ спецификатора. Поддерживаемые спецификаторы перечислены в табл. 3.4. выше. Некоторые спецификаторы могут принимать один или более модификаторов, которые перечислены в табл. 3.5 выше.

Рассмотрим теперь несколько представительных примеров строк формата, чтобы получить более полное представление о том, как они действуют. В каждом примере будет сначала демонстрироваться небольшой фрагмент программного кода, а затем производимый им вывод¹.

¹ Строки формата, используемые в языке Go, будут знакомы программистам на C, C++ и Python 2, хотя и с некоторыми малозаметными

3.5.1. Форматирование логических значений

Логические значения выводятся с помощью спецификатора формата `%t`.

```
fmt.Printf("%t %t\n", true, false)
true false
```

Если потребуется вывести логическое значение как целое число, его необходимо будет сначала вручную преобразовать в число:

```
fmt.Printf("%d %d\n", IntForBool(true), IntForBool(false))
1 0
```

В этой инструкции используется следующая пользовательская функция.

```
func IntForBool(b bool) int {
    if b {
        return 1
    }
    return 0
}
```

Выполнить обратное преобразование строки в логическое значение можно с помощью функции `strconv.ParseBool()`. И, разумеется, существуют аналогичные функции преобразования строк в числа (§3.6.2).

3.5.2. Форматирование целочисленных значений

Теперь посмотрим, как формируются *целочисленные значения*, начав с двоичного представления (в системе счисления по основанию 2).

```
fmt.Printf("|%b|%9b|%-9b|%09b|% 9b|\n", 37, 37, 37, 37, 37)
|100101|...100101|100101...|000100101|...100101|
```

Первое определение формата (`%b`) содержит спецификатор `%b` (binary — двоичный) и обеспечивает вывод целого числа в

отличиями. Например, спецификатор `%d` в языке Go можно применять для вывода значений любых целочисленных типов независимо от их размера или наличия знака.

двоичном представлении с использованием как можно меньшего количества цифр. Второе определение формата (%9b) задает ширину поля вывода, равную 9 символам (ширина поля может быть увеличена, если это будет необходимо, чтобы избежать усечения числа) и обеспечивает выравнивание по правому краю. Третье определение формата (%-9b) содержит модификатор -, обеспечивающий выравнивание по левому краю. Четвертое определение формата (%09b) указывает, что дополнение слева до полной ширины поля должно производиться символом 0, и пятое определение формата (% 9b) указывает, что дополнение должно выполняться символами пробела.

Вывод целых чисел в восьмеричном виде реализуется аналогично выводу в двоичном виде, но, кроме того, поддерживается альтернативный формат вывода. Для этой цели используется спецификатор формата %o (octal – восьмеричный).

```
fmt.Printf("|%o|%#o|%# 8o|%#+ 8o|%+08o|\n", 41, 41, 41, 41, -41)
|51|051|.....051|....+051|-0000051|
```

Альтернативный формат вывода включается модификатором # и заключается в выводе ведущего нуля перед числом. Модификатор + обеспечивает принудительный вывод знака числа – без него положительные числа выводятся без знака.

Для вывода целых чисел в шестнадцатеричном виде применяются спецификаторы %x и %X (hexadecimal – шестнадцатеричный) – они отличаются использованием символов нижнего или верхнего регистра для цифр A–F.

```
i := 3931
fmt.Printf("|%x|%X|%8x|%8X|%#04X|0x%04X|\n", i, i, i, i, i, i)
|f5b|F5B|.....f5b|00000f5b|0X0F5B|0x0F5B|
```

Для шестнадцатеричных спецификаторов модификатор альтернативного формата (#) обеспечивает вывод префикса 0x или 0X. Как и для любых других чисел, если указанная в спецификаторе ширина поля больше, чем необходимо для вывода числа, слева добавляются дополнительные пробелы, чтобы обеспечить вывод числа в поле заданной ширины с выравниванием по правому краю, а если ширина поля оказывается слишком маленькой, число будет выведено целиком, чтобы исключить усечение значимых цифр.

Для вывода целых чисел в десятичном виде используется спецификатор %d (decimal – десятичный). Дополнением слева до ширины поля может использоваться только пробел или ноль, однако легко реализовать дополнение любыми другими символами с помощью собственной функции.

```
i = 569
fmt.Printf("%d|%06d|$+06d|$s|\n", i, i, i, Pad(i, 6, '*'))
|569|$000569|$+00569|$***569|
```

В последнем определении формата используется спецификатор %s (string – строка) для вывода строки, возвращаемой функцией Pad().

```
func Pad(number, width int, pad rune) string {
    s := fmt.Sprintf(number)
    gap := width - utf8.RuneCountInString(s)
    if gap > 0 {
        return strings.Repeat(string(pad), gap) + s
    }
    return s
}
```

Функция `utf8.RuneCountInString()` возвращает количество символов в указанной строке. Это число всегда меньше или равно числу байтов. Функция `strings.Repeat()` принимает строку и счетчик, возвращает новую строку, содержащую указанную строку, повторяющуюся заданное число раз. Здесь символ, используемый для дополнения, передается в виде значения типа `rune` (то есть в виде значения кодового пункта Юникода), чтобы исключить возможность передачи строк, содержащих более одного символа.

3.5.3. Форматирование символов

Символы в языке Go представлены типом `rune` (то есть `int32`) и могут выводиться как числа или как символы Юникода.

```
fmt.Printf("%d %#04x %U '%c'\n", 0x3A6, 934, '\u03A6', '\U000003A6')
934·0x03a6·U+03A6·'Φ'
```

Здесь заглавная буква *Phi* («Φ») греческого алфавита выводится в виде десятичного и шестнадцатеричного чисел, а также в виде кодового пункта Юникода – с использованием спецификатора %U

(Unicode – Юникод) и символа Юникода – с использованием спецификатора `%c` (character (символ) или code point (кодировый пункт)).

3.5.4. Форматирование вещественных значений

При выводе *вещественных чисел* можно указать общую ширину поля вывода, количество цифр после десятичной точки и форму представления – стандартную или научную.

```
for _, x := range []float64{-2.258, 7194.84, -60897162.0218, 1.500089e-8} {
    fmt.Printf("%20.5e|%20.5f|%s|\n", x, x, Humanize(x, 20, 5, '*', ','))
}
|.....-2.58000e-01|.....-0.25800|*****-0.25800|
|.....7.19484e+03|.....7194.84000|*****7,194.84000|
|.....-6.08972e+07|.....-60897162.02180|***-60,897,162.02180|
|.....1.50009e-08|.....0.00000|*****0.00000|
```

Для итераций по значениям типа `float64` в литерале среза здесь использован цикл `for ...range`.

Пользовательская функция `Humanize()` возвращает строковое представление числа с разделителями групп разрядов (для стран, где используется группировка по три разряда) и дополненное слева указанным символом.

```
func Humanize(amount float64, width, decimals int,
    pad, separator rune) string {
    dollars, cents := math.Modf(amount)
    whole := fmt.Sprintf("%.0f", dollars)[1:] // Отбросить "±"
    fraction := ""
    if decimals > 0 {
        fraction = fmt.Sprintf("%.*f", decimals, cents)[2:] // Отбросить "±0"
    }
    sep := string(separator)
    for i := len(whole) - 3; i > 0; i -= 3 {
        whole = whole[:i] + sep + whole[i:]
    }
    if amount < 0.0 {
        whole = "-" + whole
    }
    number := whole + fraction
    gap := width - utf8.RuneCountInString(number)
    if gap > 0 {
```

```

    return strings.Repeat(string(pad), gap) + number
}
return number
}

```

Функция `math.Modf()` возвращает целую и дробную части числа типа `float64` в виде двух значений `float64`. Чтобы получить представление целой части в виде строки, была использована функция `fmt.Sprintf()` со спецификатором формата, обеспечивающим принудительный вывод знака, который тут же отбрасывается с помощью операции извлечения среза строки. Аналогичный прием применяется к дробной части, только на этот раз количество знаков после десятичной точки было указано равным значению аргумента `decimals` с помощью модификатора формата `.m`, как `*`. (То есть если в аргументе `decimals` передать значение 2, определение формата фактически примет вид `%+.2f`.) Из строкового представления дробной части удаляются начальные символы `-0` или `+0`.

Разделитель групп разрядов вставляется в строку, представляющую целую часть числа, в направлении справа налево, а затем, если число отрицательное, добавляется знак «-». В конце целая и дробная части объединяются, и результат возвращается вызывающей программе, дополняясь слева, если необходимо.

Спецификаторы формата `%e`, `%E`, `%f`, `%g` и `%G` могут применяться не только к вещественным, но и к комплексным числам. Спецификаторы `%e` и `%E` обеспечивают вывод числа в научной (экспоненциальной) форме, спецификатор `%f` — в обычной форме представления вещественных чисел, и спецификаторы `%g` и `%G` являются универсальными спецификаторами формата для вещественных значений.

Важно помнить, что при форматировании комплексных чисел модификаторы применяются к обоим частям, действительной и мнимой по отдельности, например при применении формата `%6f` к комплексному числу получившаяся строка будет содержать не менее 20 символов.

```

for _, x := range []complex128{2 + 3i, 172.6 - 58.3019i,
    -.827e2 + 9.04831e-3i} {
    fmt.Printf("%15s|%9.3f|%%.2f|%.1e|\n",
        fmt.Sprintf("%.6.2f%+.3fi", real(x), imag(x)), x, x, x)
}
|...2.00+3.000i|(...2.000...+3.000i)|(2.00+3.00i)|(2.0e+00+3.0e+00i)|
|172.60-58.302i|(.172.600...-58.302i)|(172.60-58.30i)|(1.7e+02-5.8e+01i)|
|...-82.70+0.009i|(...-82.700...+0.009i)|(-82.70+0.01i)|(-8.3e+01+9.0e-03i)|

```

При выводе необходимо было, чтобы в первом столбце компоненты комплексного числа выводились с разным количеством цифр после десятичной точки. Для этого действительная и мнимая части форматируются отдельно, с помощью функции `fmt.Sprintf()`, и затем полученная строка выводится с использованием формата `%15s`. Для других столбцов использовались непосредственно спецификаторы `%f` и `%e` – они всегда заключают комплексные числа в круглые скобки.

3.5.5. Форматирование строк и срезов

При выводе *строк* можно указать минимальную ширину поля вывода (слишком короткие строки будут дополняться пробелами) и максимальное количество выводимых символов (слишком длинные строки будут усекаться). Строки могут выводиться как текст Юникода (то есть как последовательности символов) или как последовательности кодовых пунктов (то есть значений типа `rune`) или байтов UTF-8, представляющих их.

```
slogan := "End Óréttlæti♥"
fmt.Printf("%s\n%q\n%+q\n%#q\n", slogan, slogan, slogan, slogan)
End Óréttlæti♥
"End Óréttlæti♥"
"End \u00d3r\u00e9ttl\u00e6ti\u2665"
`End Óréttlæti♥`
```

Для вывода строк используется спецификатор `%s` – мы вернемся к нему чуть ниже. Для вывода строк в двойных кавычках используется спецификатор `%q` (quoted string – строка в кавычках), который обеспечивает вывод печатаемых символов буквально, а всех остальных – в виде экранированных последовательностей (см. табл. 3.1 выше). Если добавить модификатор `+`, буквально будут выводиться только ASCII-символы (в диапазоне от `U+0020` до `U+007E`), а все остальные – в виде экранированных последовательностей. Если добавить модификатор `#`, будут выводиться строки в обратных апострофах, если возможно, в противном случае – в кавычках.

Обычно в программах спецификаторам формата соответствуют переменные с единственным значением совместимого типа (например, типа `int` для спецификатора `%d` или `%x`), однако переменная может также содержать срез или отображение, в котором тип ключей и значений совместим со спецификатором (например, когда и ключи, и значения являются строками или числами).

```
chars := []rune(slogan)
fmt.Printf("%x\n%x\n%X\n", chars, chars, chars)
[45·6e·64·20·d3·72·e9·74·74·6c·e6·74·69·2665]
[0x45·0x6e·0x64·0x20·0xd3·0x72·0xe9·0x74·0x74·0x6c·0xe6·0x74·0x69·0x2665]
[0X45·0X6E·0X64·0X20·0XD3·0X72·0XE9·0X74·0X74·0X6C·0XE6·0X74·0X69·0X2665]
```

Здесь выводится срез со значениями типа `rune` — кодовых пунктов, в данном примере — в виде последовательности шестнадцатеричных чисел, по одному на кодовый пункт, с использованием спецификаторов `%x` и `%X`. Наличие модификатора `#` обеспечивает принудительный вывод начальных символов `0x` или `0X` перед каждым числом.

Для большинства типов срезы выводятся как последовательности их элементов, разделенных пробелами, заключенными в квадратные скобки. Исключение составляет тип `[]byte`, для которого квадратные скобки и пробелы выводятся только при использовании спецификатора `%v`.

```
bytes := []byte(slogan)
fmt.Printf("%s\n%x\n%X\n X\n%v\n", bytes, bytes, bytes, bytes, bytes)
End·Öréttlæti♥
456e6420c39372c3a974746cc3a67469e299a5
456E6420C39372C3A974746CC3A67469E299A5
45·6E·64·20·C3·93·72·C3·A9·74·74·6C·C3·A6·74·69·E2·99·A5
[69·110·100·32·195·147·114·195·169·116·116·108·195·166·116·105·226·153·165]
```

Срез, содержащий элементы типа `bytes`, в данном случае байты UTF-8, представляющие строку, можно вывести как последовательность двухразрядных шестнадцатеричных чисел, по одному на байт. При использовании спецификатора `%s` байты интерпретируются как символы Юникода в кодировке UTF-8 и выводятся как строка. Для срезов типа `[]bytes` не существует альтернативного шестнадцатеричного формата вывода, но числа могут быть отделены друг от друга пробелами, как видно в последней строке вывода. Спецификатор `%v` выводит срезы типа `[]bytes` как последовательности десятичных чисел, разделенных пробелами и заключенных в скобки.

По умолчанию в языке Go используется *выравнивание по правому краю*. Чтобы обеспечить *выравнивание по левому краю*, можно воспользоваться модификатором `-`. И конечно, можно указать минимальную ширину поля вывода и *максимальное число символов*, как показано в следующих двух примерах.

```
s := "Dare to be naïve"
fmt.Printf("|%22s|%-22s|%10s|\n", s, s, s)
|.....Dare.to.be.naïve|Dare.to.be.naïve.....|Dare.to.be.naïve|
```

В этом фрагменте в третьем определении формата (%10s) задается минимальная ширина поля, равная 10 символам, но, так как выводимая строка длиннее этого значения, а само значение определяет минимальную ширину поля, строка выводится полностью.

```
i := strings.Index(s, "n")
fmt.Printf("|%10s|%. *s|%-22.10s|%s|\n", s, i, s, s, s)
|Dare.to.be|Dare.to.be.|Dare.to.be.....|Dare.to.be.naïve|
```

Здесь первое определение формата (%.10s) указывает, что выведено может быть не более 10 символов из строки, поэтому строка усекается в соответствии с заданным значением. Второе определение формата (%.*s) предполагает наличие двух аргументов – значения, определяющего максимальное число символов, и строки. Здесь в качестве максимального количества использован номер позиции символа n, поэтому выводятся только символы, стоящие перед ним, не включая самого этого символа. Третье определение формата (%-22.10s) задает минимальную ширину поля вывода, равную 22 символам, и максимальное число символов для вывода, равное 10. В этом случае выводятся только первые 10 символов из исходной строки, но в поле шириной 22 символа. Поскольку поле оказалось шире, чем число выводимых символов, оно было дополнено пробелами, причем справа, потому что был использован модификатор – выравнивания по левому краю.

3.5.6. Форматирование для отладки

Спецификатор %T (type – тип) используется для вывода информации о типе значения, а спецификатор %v – для вывода значения встроенного типа. В действительности спецификатор %v можно также применять для вывода значений *пользовательских типов* – в этом случае вызывается метод String() типа, а для типов, не имеющих его, используется формат вывода по умолчанию.

```
p := polar{-83.40, 71.60}
fmt.Printf("|%T|%v|%#v|\n", p, p, p)
fmt.Printf("|%T|%v|%t|\n", false, false, false)
```

```
fmt.Printf("|%T|%v|%d|\n", 7607, 7607, 7607)
fmt.Printf("|%T|%v|%f|\n", math.E, math.E, math.E)
fmt.Printf("|%T|%v|%f|\n", 5+7i, 5+7i, 5+7i)
s := "Relativity"
fmt.Printf("|%T|"%v\`|`"%s\`|%q|\n", s, s, s, s)
|main.polar|{-83.4·71.6}|main.polar{radius:-83.4,·q:71.6}| |
|bool|false|false|
|int|7607|7607|
|float64|2.718281828459045|2.718282|
|complex128|(5+7i)|(5.000000+7.000000i)|
|string|"Relativity"|"Relativity"|"Relativity"|
```

Этот пример демонстрирует, как выводятся информация о типе и значение произвольного типа с помощью спецификаторов `%T` и `%v`. Если формат вывода, обеспечиваемый спецификатором `%v`, удовлетворяет требованиям, для вывода можно использовать простую функцию `fmt.Print()` и аналогичные ей, так как по умолчанию они задействуют спецификатор формата `%v`. Добавление модификатора `#` альтернативного формата в спецификатор `%v` оказывает влияние только на вывод структур – в этом случае выводятся названия типов структур и имена полей. Для вещественных значений спецификатор `%v` действует подобно спецификатору `%g`, а не `%f`. Спецификатор `%T` главным образом применяется при отладке – для пользовательских типов он включает в вывод имя пакета (в данном случае `main`). При применении спецификатора `%q` к строкам он заключает их в кавычки, что также часто бывает удобно при отладке.

В языке Go имеются два типа, являющиеся синонимами для двух других типов: `byte` – для типа `uint8` и `rune` – для типа `int32`. Тип `int32` следует использовать для обработки 32-битных целых чисел со знаком, где использование типа `int` может оказаться неуместным (например, при работе с двоичными файлами), а тип `rune` – для работы с кодовыми пунктами Юникода (символами).

```
s := "Alias↔Synonym"
chars := []rune(s)
bytes := []byte(s)
fmt.Printf("%T: %v\n%T: %v\n", chars, chars, bytes, bytes)
[]int32: [65 108 105 97 115 8596 83 121 110 111 110 121 109]
[]uint8: [65 108 105 97 115 226 134 148 83 121 110 111 110 121 109]
```

Как иллюстрирует этот фрагмент, спецификатор `%T` всегда выводит оригинальное имя типа, а не его синоним. Так как строка

содержит не ASCII-символ, очевидно, что в этом фрагменте создаются срез, содержащий значения типа `rune` (кодировые пункты), и срез, содержащий байты в кодировке UTF-8.

В языке Go имеется также возможность с помощью спецификатора `%p` (`pointer` – указатель) выводить адреса значений в памяти.

```
i := 5
f := -48.3124
s := "Tomás Bretón"
fmt.Printf("|%p → %d|p → %f|p → %s|\n", &i, i, &f, f, &s, s)
|0xf840000300 → 5|0xf840000308 → -48.312400|f840001990 → Tomás·Bretón|
```

Оператор `&` получения адреса описывается в следующей главе (§4.1). Если в спецификатор `%p` добавить модификатор `#`, начальные символы `0x` в адресе выводиться не будут. Вывод адресов в памяти, как в данном примере, может пригодиться при отладке.

Также может оказаться полезной при отладке возможность вывода содержимого срезов и отображений, как и возможность вывода значений каналов – то есть значений, передаваемых и принимаемых посредством канала, и адрес в памяти самого канала.

```
fmt.Println([]float64{math.E, math.Pi, math.Phi})
fmt.Printf("%v\n", []float64{math.E, math.Pi, math.Phi})
fmt.Printf("%#v\n", []float64{math.E, math.Pi, math.Phi})
fmt.Printf("%.5f\n", []float64{math.E, math.Pi, math.Phi})
[2.718281828459045·3.141592653589793·1.618033988749895]
[2.718281828459045·3.141592653589793·1.618033988749895]
[]float64{2.718281828459045, 3.141592653589793, 1.618033988749895}
[2.71828·3.14159·1.61803]
```

При применении немодифицированного спецификатора `%v` к срезам они выводятся в виде заключенной в квадратные скобки последовательности элементов, разделенных пробелами. Обычно срезы выводятся с помощью таких функций, как `fmt.Print()` или `fmt.Sprint()`, но когда используется функция форматированного вывода, чаще применяется спецификатор `%v` или `%#v`. Однако имеется возможность использовать спецификаторы, совместимые с типом элементов, такие как `%f` – для вещественных чисел или `%s` – для строк.

```
fmt.Printf("%q\n", []string{"Software patents", "kill", "innovation"})
fmt.Printf("%v\n", []string{"Software patents", "kill", "innovation"})
fmt.Printf("%#v\n", []string{"Software patents", "kill", "innovation"})
```

```
fmt.Printf("%17s\n", []string{"Software patents", "kill", "innovation"})
["Software.patents"."kill"."innovation"]
[Software.patents.kill.innovation]
[]string{"Software.patents", ".kill", ".innovation"}
[.Software.patents.....kill.....innovation]
```

Спецификатор `%q` особенно удобно использовать для вывода срезов со строками, содержащих пробелы, так как это позволяет визуально отделить строки друг от друга, чего не скажешь о спецификаторе `%v`.

Последняя строка в выводе выше может показаться ошибочной, так как она содержит 53 символа (не включая квадратных скобок), а не 51 (три строки по 17 символов, ни одна из которых не превышает ширины поля вывода). Очевидное несоответствие обусловлено наличием разделительных пробелов, которые выводятся между элементами среза.

Кроме того, спецификатор `%#v` может оказаться полезным при отладке, когда код на языке Go генерируется программно.

```
fmt.Printf("%v\n", map[int]string{1: "A", 2: "B", 3: "C", 4: "D"})
fmt.Printf("%#v\n", map[int]string{1: "A", 2: "B", 3: "C", 4: "D"})
fmt.Printf("%v\n", map[int]int{1: 1, 2: 2, 3: 4, 4: 8})
fmt.Printf("%#v\n", map[int]int{1: 1, 2: 2, 3: 4, 4: 8})
fmt.Printf("%04b\n", map[int]int{1: 1, 2: 2, 3: 4, 4: 8})
map[4:D·1:A·2:B·3:C]
map[int]·string{4:"D", ·1:"A", ·2:"B", ·3:"C"}
map[4:8·1:1·2:2·3:4]
map[int]·int{4:8, ·1:1, ·2:2, ·3:4}
map[0100:1000·0001:0001·0010:0010·0011:0100]
```

Отображения выводятся как слово «map», за которым следуют пары *ключ/значение* (в произвольном порядке, потому что отображения являются неупорядоченными последовательностями). Как и в случае со срезами, для вывода содержимого *отображений* можно использовать спецификаторы, отличные от `%v`, но только если тип ключей и значений совместим с используемым спецификатором, как в последней инструкции в примере выше. (Отображения и срезы будут подробно рассматриваться в главе 4.)

Функции вывода в пакете `fmt` обладают большой гибкостью и могут использоваться для вывода любых значений. Единственное, чего не позволяют эти функции, — использования произвольных

символы для дополнения строк до указанной ширины поля (помимо нуля и пробела). Но, как было показано на примере функций `Pad()` и `Humanize()` (§3.5.2), такую возможность легко реализовать самому.

3.6. Другие пакеты для работы со строками

Поддержка строк в языке Go не ограничивается операциями индексирования и извлечения срезов и гибкими функциями в пакете `fmt`. В частности, богатейший набор функций содержится в пакете `strings`. Пакеты `strconv`, `unicode/utf8` и `unicode` также содержат массу полезных функций. В этом разделе будут представлены примеры использования функций из всех этих пакетов. В нескольких примерах на протяжении книги используется поддержка регулярных выражений, реализованная в виде мощного пакета `regexp`, который будет представлен далее в этом разделе.

В стандартной библиотеке имеются и другие пакеты, предоставляющие функции для работы со строками. Некоторые из них охватываются в разных главах в книге, в примерах или упражнениях.

3.6.1. Пакет *strings*

Обычно при работе со строками бывает необходимо разбивать их на отдельные подстроки для дальнейшей обработки. Например, чтобы преобразовать строки в числа или чтобы удалить пробельные символы на концах строк.

Чтобы вы могли получить представление, как используются некоторые функции из пакета `strings`, ниже представлено несколько коротких примеров. Все функции, имеющиеся в пакете, перечислены в табл. 3.6 и в 3.7 (ниже). Начнем с примера разбиения строк.

```
names := "Niccolò•Noël•Geoffrey•Amélie••Turlough•José"
fmt.Print("«|»")
for _, name := range strings.Split(names, "•") {
    fmt.Printf("%s|", name)
}
fmt.Println()
|Niccolò|Noël|Geoffrey|Amélie||Turlough|José|
```

Здесь имеется список имен, разделенных маркерами (включая один пустой элемент), который разбивается с помощью функции `strings`.

`Split()`. Эта функция принимает исходную строку и строку-разделитель, по которой должно выполняться разбиение, и делит исходную строку на максимально возможное количество фрагментов. (Если потребуется ограничить количество разбиений, можно воспользоваться функцией `strings.SplitN()`.) При использовании функции `strings.SplitAfter()` фрагмент выше вывел бы следующую строку:

```
|Niccolò•|Noël•|Geoffrey•|Amélie••|Turlough•|José|
```

Функция `strings.SplitAfter()` разбивает исходную строку точно так же, как и функция `strings.Split()`, но сохраняет строку-разделитель. Существует также функция `strings.SplitAfterN()` на случай, если потребуется ограничить число разбиений.

Для разбиения строк по любому из двух или более различных символов можно использовать функцию `strings.FieldsFunc()`.

```
for _, record := range []string{"László Lajtha•1892•1963",
    "Édouard Lalo\t1823\t1892", "José Ángel Lamas|1775|1814"} {
    fmt.Println(strings.FieldsFunc(record, func(char rune) bool {
        switch char {
        case '\t', '•', '|':
            return true
        }
        return false
    })))
}
[László•Lajtha•1892•1963]
[Édouard•Lalo•1823•1892]
[José•Ángel•Lamas•1775•1814]
```

Функция `strings.FieldsFunc()` принимает строку (в данном примере – переменную `record`) и ссылку на функцию с сигнатурой `func(rune) bool`. Поскольку функция достаточно маленькая и используется только в одном месте, она была определена как *анонимная функция* в месте, где она используется. (Функции, созданные таким способом, образуют *замыкания*, хотя в данном случае замкнутое состояние не используется (§5.6.3).) Функция `strings.FieldsFunc()` выполняет итерации по всем символам в указанной строке, для каждого из них вызывает функцию, переданную во втором аргументе, и выполняет разбиение, если вызванная функция вернет `true`. В данном случае строка разбивается по символам табуляции, звездочки и вертикальной черты. (Инструкция `switch` в языке Go рассматривается в §5.2.2.)

Таблица 3.6. Функции из пакета *strings*, часть 1

Переменные *s* и *t* имеют тип `string`, *xs* – тип `[]string`, *i* – тип `int`, и *f* – ссылка на функцию с сигнатурой `func(rune) bool`. Индексы соответствуют первым байтам в кодировке UTF-8 для кодовых пунктов Юникода (символов) или строк, или имеют значение `-1` в случае отсутствия соответствия.

Функция	Описание/результат
<code>strings.Contains(s, t)</code>	<code>true</code> , если подстрока <i>t</i> входит в строку <i>s</i>
<code>strings.Count(s, t)</code>	Количество (неперекрывающихся) вхождений подстроки <i>t</i> в строку <i>s</i>
<code>strings.EqualFold(s, t)</code>	<code>true</code> , если строки равны, без учета регистра символов
<code>strings.Fields(s)</code>	Срез <code>[]string</code> , содержащий результаты разбиения строки <i>s</i> по пробельным символам
<code>strings.FieldsFunc(s, f)</code>	Срез <code>[]string</code> , содержащий результаты разбиения строки <i>s</i> по позициям, для которых функция <i>f</i> вернет <code>true</code>
<code>strings.HasPrefix(s, t)</code>	<code>true</code> , если строка <i>s</i> начинается со строки <i>t</i>
<code>strings.HasSuffix(s, t)</code>	<code>true</code> , если строка <i>s</i> заканчивается строкой <i>t</i>
<code>strings.Index(s, t)</code>	Индекс первого вхождения подстроки <i>t</i> в строке <i>s</i>
<code>strings.IndexAny(s, t)</code>	Индекс первого вхождения любого символа из строки <i>t</i> в строке <i>s</i>
<code>strings.IndexFunc(s, f)</code>	Индекс первого вхождения символа в строке <i>s</i> , для которого функция <i>f</i> вернет <code>true</code>
<code>strings.IndexRune(s, char)</code>	Индекс первого вхождения символа <code>char</code> , представленного значением типа <code>rune</code> , в строке <i>s</i>
<code>strings.Join(xs, t)</code>	Строка, являющаяся результатом конкатенации всех строк в <i>xs</i> , разделенных строкой <i>t</i> (которая может быть пустой строкой)
<code>strings.LastIndex(s, t)</code>	Индекс последнего вхождения подстроки <i>t</i> в строке <i>s</i>
<code>strings.LastIndexAny(s, t)</code>	Индекс последнего вхождения любого символа из строки <i>t</i> в строке <i>s</i>
<code>strings.LastIndexFunc(s, f)</code>	Индекс последнего вхождения символа в строке <i>s</i> , для которого функция <i>f</i> вернет <code>true</code>
<code>strings.Map(mf, t)</code>	Копия строки <i>t</i> , в которой каждый символ замещается или удаляется в соответствии с функцией отображения <i>mf</i> , имеющей сигнатуру <code>func(rune) rune</code> (см. описание в тексте)
<code>strings.NewReader(s)</code>	Указатель на значение с методами <code>Read()</code> , <code>ReadByte()</code> и <code>ReadRune()</code> для выполнения операций со строкой <i>s</i>
<code>strings.NewReplacer(...)</code>	Указатель на значение с методами для замены каждой указанной пары строк старая/новая
<code>strings.Repeat(s, i)</code>	Строка, в которой строка <i>s</i> повторяется <i>i</i> раз

Таблица 3.7. Функции из пакета *strings*, часть 2

Переменная `r` имеет тип `unicode.SpecialCase` и используются для определения дополнительных правил Юникода.

Функция	Описание/результат
<code>strings.Replace(s, old, new, i)</code>	Копия строки <code>s</code> , где все неперекрывающиеся вхождения строки <code>old</code> замещены строкой <code>new</code> , если в аргументе <code>i</code> передано значение <code>-1</code> . В противном случае выполняется не более <code>i</code> замен
<code>strings.Split(s, t)</code>	Срез <code>[]string</code> , являющийся результатом разбиения строки <code>s</code> по вхождениям строки <code>t</code>
<code>strings.SplitAfter(s, t)</code>	Действует подобно функции <code>strings.Split()</code> , но сохраняет строку-разделитель в полученных строках (см. описание в тексте)
<code>strings.SplitAfterN(s, t, i)</code>	Действует подобно функции <code>strings.SplitN()</code> , но сохраняет строку-разделитель в полученных строках (см. описание в тексте)
<code>strings.SplitN(s, t, i)</code>	Срез <code>[]string</code> , являющийся результатом разбиения строки <code>s</code> по вхождениям строки <code>t</code> , но не более <code>i-1</code> раз
<code>strings.Title(s)</code>	Копия строки <code>s</code> , где первые буквы в каждом слове преобразованы в верхний регистр
<code>strings.ToLower(s)</code>	Копия строки <code>s</code> , где все символы преобразованы в нижний регистр
<code>strings.ToLowerSpecial(r, s)</code>	Копия строки <code>s</code> , где все символы преобразованы в нижний регистр, при этом предпочтение отдается правилам в <code>r</code>
<code>strings.ToTitle(s)</code>	Копия строки <code>s</code> , где первые символы слов преобразованы в верхний регистр
<code>strings.ToTitleSpecial(r, s)</code>	Копия строки <code>s</code> , где первые символы слов преобразованы в верхний регистр, при этом предпочтение отдается правилам в <code>r</code>
<code>strings.ToUpper(s)</code>	Копия строки <code>s</code> , где все символы преобразованы в верхний регистр
<code>strings.ToUpperSpecial(r, s)</code>	Копия строки <code>s</code> , где все символы преобразованы в верхний регистр, при этом предпочтение отдается правилам в <code>r</code>
<code>strings.Trim(s, t)</code>	Копия строки <code>s</code> , где с обоих концов удалены символы, входящие в строку <code>t</code>
<code>strings.TrimFunc(s, f)</code>	Копия строки <code>s</code> , где с обоих концов удалены символы, для которых функция <code>f</code> вернула <code>true</code>
<code>strings.TrimLeft(s, t)</code>	Копия строки <code>s</code> , в начале которой удалены символы, входящие в строку <code>t</code>

Таблица 3.7. Функции из пакета *strings*, часть 2

Функция	Описание/результат
<code>strings.TrimLeftFunc(s, f)</code>	Копия строки <i>s</i> , где в начале удалены символы, для которых функция <i>f</i> вернула <code>true</code>
<code>strings.TrimRight(s, t)</code>	Копия строки <i>s</i> , где в конце удалены символы, входящие в строку <i>t</i>
<code>strings.TrimRightFunc(s, f)</code>	Копия строки <i>s</i> , где в конце удалены символы, для которых функция <i>f</i> вернула <code>true</code>
<code>strings.TrimSpace(s)</code>	Копия строки <i>s</i> , где с обоих концов удалены пробельные символы

Замену всех вхождений подстроки в строке можно выполнить с помощью функции `strings.Replace()`. Например:

```
names = " Antônio\tAndré\tFriedrich\t\tJean\t\tÉlisabeth\tIsabella \t"
names = strings.Replace(names, "\t", " ", -1)
fmt.Printf("|%s|\n", names)
|·Antônio·André·Friedrich··Jean·Élisabeth·Isabella·|
```

Функция `strings.Replace()` принимает строку для обработки, искомую подстроку, строку замены, количество замен (значение `-1` означает максимально возможное количество замен) и возвращает строку, где все вхождения (неперекрывающиеся) искомой подстроки замещены указанной строкой замены.

Строки, введенные пользователем или полученные из внешних источников, часто бывает необходимо нормализовать по пробельным символам: то есть удалить пробельные символы в начале и в конце строки и заменить последовательности пробельных символов внутри строки единственным пробелом.

```
fmt.Printf("|%s|\n", SimpleSimplifyWhitespace(names))
|Antônio·André·Friedrich·Jean·Élisabeth·Isabella|
```

Ниже приводится реализация однострочной функции `SimpleSimplifyWhitespace()`.

```
func SimpleSimplifyWhitespace(s string) string {
    return strings.Join(strings.Fields(strings.TrimSpace(s)), " ")
}
```

Функция `strings.TrimSpace()` возвращает копию переданной ей строки, в которой удалены все начальные и конечные пробельные

символы. Функция `strings.Fields()` разбивает исходную строку по пробельным символам и возвращает значение типа `[]string`. А функция `strings.Join()` принимает значение типа `[]string`, строку-разделитель (может быть пустой строкой, однако здесь используется пробел) и возвращает единую строку, в которую через строку-разделитель объединены все строки из среза типа `[]string`. С помощью комбинации этих трех функций выполняется *нормализация строк по пробельным символам*.

Разумеется, обработку пробельных символов можно выполнить более эффективным способом, воспользовавшись типом `bytes.Buffer`.

```
func SimplifyWhitespace(s string) string {
    var buffer bytes.Buffer
    skip := true
    for _, char := range s {
        if unicode.IsSpace(char) {
            if !skip {
                buffer.WriteRune(' ')
                skip = true
            }
        } else {
            buffer.WriteRune(char)
            skip = false
        }
    }
    s = buffer.String()
    if skip && len(s) > 0 {
        s = s[:len(s)-1]
    }
    return s
}
```

Функция `SimplifyWhitespace()` выполняет итерации по символам в переданной ей строке, пропускает начальные пробельные символы, используя для этого функцию `unicode.IsSpace()` (см. табл. 3.11 ниже). Затем накапливает символы, записывая их в значение типа `bytes.Buffer`, замещая все последовательности из одного или более пробельных символов единственным пробелом. В конце удаляется завершающий пробел (в соответствии с алгоритмом к данному моменту в конце строки может содержаться не более одного пробела) и результат возвращается вызывающей программе. Намного проще

выглядит версия, использующая регулярные выражения (которая будет показана ниже).

Функция `strings.Map()` может применяться для замены или удаления символов из строк. Она принимает два аргумента: первый – функция отображения с сигнатурой `func(rune) rune` и второй – строка. Функция отображения вызывается для каждого символа в исходной строке, и каждый символ замещается символом, возвращаемым функцией, или удаляется, если функция отображения вернет отрицательное число.

```
asciiOnly := func(char rune) rune {  
    if char > 127 {  
        return '?'  
    }  
    return char  
}  
  
fmt.Println(strings.Map(asciiOnly, "Jérôme Österreich"))  
J?r?me.?sterreich
```

Здесь, в отличие от примера использования функции `strings.FieldsFunc()` (выше), функция отображения создается не в месте ее использования, а определяется как *анонимная функция*, и ссылка на нее присваивается переменной (`asciiOnly`). Затем вызывается функция `strings.Map()`, которой передаются переменная со ссылкой на функцию отображения и строка для обработки. В результате получается строка, где все не ASCII-символы замещены символом «?». Разумеется, функцию отображения можно было бы определить в месте ее использования, но подобная организация программного кода, как в данном примере, более удобна, когда функция отображения получается достаточно длинной или используется многократно.

Подобный подход можно использовать для удаления не ASCII-символов, чтобы, например, воспроизвести строку:

```
Jrme.sterreich
```

Для этого достаточно в функции отображения вместо символа «?» вернуть значение `-1`, если символ не является ASCII-символом.

Выше упоминалось, что имеется возможность с помощью цикла `for ... range (§5.3)` выполнять итерации по символам (кодovým пун-ктам Юникода) в строке. Аналогичного эффекта можно добиться

при чтении данных из типов, реализующих функцию `ReadRune()`, таких как `bufio.Reader`.

```
for {
    char, size, err := reader.ReadRune()
    if err != nil {           // ошибка может возникать при чтении из файла
        if err == io.EOF { // завершить обработку как обычно
            break
        }
        panic(err)          // прервать выполнение по ошибке
    }
    fmt.Printf("%U '%c' %d: % X\n", char, char, size, []byte(string(char)))
}
U+0043·'C'·1:·43
U+0061·'a'·1:·61
U+0066·'f'·1:·66
U+00E9·'é'·2:·C3·A9
```

Этот фрагмент читает строку и выводит кодовый пункт каждого символа, сам символ, количество байтов и собственно байты в кодировке UTF-8, представляющие символ. В большинстве случаев значения, реализующие интерфейс `bufio.Reader`, используются для работы с файлами, поэтому в данном примере можно было бы представить, что значение переменной `reader` было создано вызовом функции `bufio.NewReader()`, которой было передано значение, полученное от функции `os.Open()`. Подобная реализация уже встречалась в главе 1, в примере `americanise` (§1.6). Однако в данном случае значение переменной `reader` было создано для выполнения операций со строкой:

```
reader := strings.NewReader("Café")
```

Значение `*strings.Reader`, возвращаемое функцией `strings.NewReader()`, предлагает подмножество функциональных возможностей типа `bufio.Reader`, в частности оно предоставляет методы `strings.Reader.Read()`, `strings.Reader.ReadByte()`, `strings.Reader.ReadRune()`, `strings.Reader.UnreadByte()` и `strings.Reader.UnreadRune`. Возможность оперировать значениями, реализующими определенный интерфейс (например, определяющий метод `ReadRune()`), а не значениями определенных типов, является одной из самых мощных и гибких особенностей языка Go и подробно рассматривается в главе 6.

3.6.2. Пакет strconv

Пакет `strconv` содержит множество функций для преобразования строк в значения других типов и значений других типов в строки. Функции, имеющиеся в пакете, перечислены в табл. 3.8 и 3.9 (см. также описание функций ввода/вывода из пакета `fmt` в §3.5 и в §8.2.) В этом разделе будут представлены несколько показательных примеров.

Часто бывает необходимо преобразовать строковое представление значения истинности в значение типа `bool`. Сделать это можно с помощью функции `strconv.ParseBool()`.

```
for _, truth := range []string{"1", "t", "TRUE", "false", "F", "0", "5"} {
    if b, err := strconv.ParseBool(truth); err != nil {
        fmt.Printf("\n{%-v}", err)
    } else {
        fmt.Print(b, " ")
    }
}
fmt.Println()
true.true.true.false.false.false
{strconv.ParseBool: parsing "5": invalid syntax}
```

Таблица 3.8. Функции из пакета `strconv`, часть 1

Параметр `bs` – значение типа `[]byte`, `base` – число, определяющее основание системы счисления (от 2 до 36), `bits` – требуемый размер результата в битах (8, 16, 32, 64 или 0 – для результата типа `int`; 32 или 64 – для результата типа `float64`), и `s` – строка.

Функция	Описание/результат
<code>strconv.AppendBool(bs, b)</code>	В конец <code>bs</code> добавляется строка «true» или «false» в зависимости от значения <code>b</code>
<code>strconv.AppendFloat(bs, f, fmt, prec, bits)</code>	В конец <code>bs</code> добавляется строка с представлением вещественного числа <code>f</code> ; назначение других параметров см. в описании функции <code>strconv.FormatFloat()</code>
<code>strconv.AppendInt(bs, i, base)</code>	В конец <code>bs</code> добавляется строка с представлением целого числа <code>i</code> типа <code>int64</code> в системе счисления по основанию <code>base</code>
<code>strconv.AppendQuote(bs, s)</code>	В конец <code>bs</code> добавляется строка <code>s</code> , обработанная функцией <code>strconv.Quote()</code>
<code>strconv.AppendQuoteRune(bs, char)</code>	В конец <code>bs</code> добавляется строка, полученная в результате обработки символа <code>char</code> функцией <code>strconv.QuoteRune()</code>

Таблица 3.8. Функции из пакета *strconv*, часть 1

Функция	Описание/результат
<code>strconv.AppendQuoteRuneToASCII(bs, char)</code>	В конец <code>bs</code> добавляется строка, полученная в результате обработки символа <code>char</code> функцией <code>strconv.QuoteRuneToASCII()</code>
<code>strconv.AppendQuoteToASCII(bs, s)</code>	В конец <code>bs</code> добавляется строка, полученная в результате обработки строки <code>s</code> функцией <code>strconv.QuoteToASCII()</code>
<code>strconv.AppendUInt(bs, u, base)</code>	В конец <code>bs</code> добавляется строка с представлением целого числа <code>u</code> типа <code>uint64</code> в системе счисления по основанию <code>base</code>
<code>strconv.Atoi(s)</code>	Строка <code>s</code> , преобразованная в значение типа <code>int</code> , и значение ошибки или <code>nil</code> ; см. также описание функции <code>strconv.ParseInt()</code>
<code>strconv.CanBackquote(s)</code>	<code>true</code> , если строка <code>s</code> может быть представлена в виде строки в обратных апострофах
<code>strconv.FormatBool(tf)</code>	Строка «true» или «false» в зависимости от значения <code>tf</code> типа <code>bool</code>
<code>strconv.FormatFloat(f, fmt, prec, bits)</code>	Значение <code>f</code> типа <code>float64</code> в виде строки. Параметр <code>fmt</code> – значение типа <code>byte</code> , представляющий спецификатор формата, ‘b’ – для %b, ‘e’ – для %e, и т. д. (см. табл. 3.4 выше). Параметр <code>prec</code> определяет количество цифр после десятичной точки, когда в параметре формата <code>fmt</code> указывается ‘e’, ‘E’ или ‘f’; или общее число цифр для формата ‘g’ или ‘G’; чтобы обеспечить вывод минимально возможного количества цифр без потери точности (как это делает функция <code>strconv.ParseFloat()</code>), в параметре <code>prec</code> можно передать значение –1. Параметр <code>bits</code> влияет на точность округления, и в нем обычно передается значение 64
<code>strconv.FormatInt(i, base)</code>	Строка с представлением целого числа <code>i</code> типа <code>int64</code> в системе счисления по основанию <code>base</code>
<code>strconv.FormatUInt(u, base)</code>	Строка с представлением целого числа <code>u</code> типа <code>uint64</code> в системе счисления по основанию <code>base</code>
<code>strconv.IsPrint(c)</code>	<code>true</code> , если параметр <code>c</code> типа <code>rune</code> является печатаемым символом
<code>strconv.Itoa(i)</code>	Строка с представлением целого числа <code>i</code> в десятичной системе счисления; см. также описание функции <code>strconv.FormatInt()</code>

Таблица 3.9. Функции из пакета *strconv*, часть 2

Параметр *bs* – значение типа `[]byte`, *base* – число, определяющее основание системы счисления (от 2 до 36), *bits* – требуемый размер результата в битах (8, 16, 32, 64 или 0 – для результата типа `int`; 32 или 64 – для результата типа `float64`), и *s* – строка.

Функция	Описание/результат
<code>strconv.ParseBool(s)</code>	<code>true</code> и <code>nil</code> , если <i>s</i> является строкой <code>"1"</code> , <code>"t"</code> , <code>"T"</code> , <code>"true"</code> , <code>"True"</code> или <code>"TRUE"</code> ; <code>false</code> и <code>nil</code> , если <i>s</i> является строкой <code>"0"</code> , <code>"f"</code> , <code>"F"</code> , <code>"false"</code> , <code>"False"</code> или <code>"FALSE"</code> ; <code>false</code> и значение ошибки в противном случае
<code>strconv.ParseFloat(s, bits)</code>	Значение типа <code>float64</code> и <code>nil</code> , если строка <i>s</i> содержит допустимое представление вещественного числа, или 0 и значение ошибки; параметр <i>bits</i> должен иметь значение 64, однако допускается передавать в нем значение 32, если требуется получить значение типа <code>float32</code>
<code>strconv.ParseInt(s, base, bits)</code>	Значение типа <code>int64</code> и <code>nil</code> , если строка <i>s</i> содержит допустимое представление целого числа, или 0 и значение ошибки; значение 0 в параметре <i>base</i> означает, что система счисления будет определена автоматически по содержимому строки <i>s</i> (наличие в начале строки последовательности символов <code>"0x"</code> или <code>"0X"</code> означает систему счисления по основанию 16, если строка начинается с символа <code>"0"</code> , это означает систему счисления по основанию 8; в остальных случаях используется десятичная система счисления), иначе будет использоваться указанная система счисления (по основанию от 2 до 36); параметр <i>bits</i> должен содержать значение 0, если в результате преобразования должно быть получено значение типа <code>int</code> , или количество бит, соответствующее требуемому целочисленному типу со знаком (например, 16 – для типа <code>int16</code>)
<code>strconv.ParseUint(s, base, bits)</code>	Значение типа <code>uint64</code> и <code>nil</code> , или 0 и значение ошибки, аналогично функции <code>strconv.ParseInt()</code> , только в результате возвращается беззнаковое целое
<code>strconv Quote(s)</code>	Значение типа <code>string</code> , содержащее представление строки <i>s</i> в кавычках с учетом синтаксиса языка Go; см. также табл. 3.1 (выше)
<code>strconv.QuoteRune(char)</code>	Значение типа <code>string</code> , содержащее представление в апострофах кодового пункта параметра <i>char</i> типа <code>rune</code> , с учетом синтаксиса языка Go

Таблица 3.9. Функции из пакета strconv, часть 2

Функция	Описание/результат
strconv. QuoteRuneToASCII(char)	Значение типа string, содержащее представление в апострофах кодового пункта параметра char типа rune, с учетом синтаксиса языка Go; не ASCII-символы представляются в виде экранированных последовательностей
strconv. QuoteToASCII(s)	Значение типа string, в кавычках с учетом синтаксиса языка Go; для представления не ASCII-символов используются экранированные последовательности
strconv.Unquote(s)	Значение типа string, содержащее представление строки s, заключенной в апострофы, кавычки или обратные апострофы, и значение ошибки
strconv.UnquoteChar(s, b)	Значение типа rune (первый символ), значение типа bool (если для представления первого символа в кодировке UTF-8 требуется более одного байта), значение типа string (остальная часть строки) и значение ошибки; если в параметре b передается символ апострофа или кавычки, соответствующие символы в строке результата будут экранированы

Все функции преобразования из пакета strconv возвращают результат преобразования и значение ошибки, которое равно nil, если преобразование было выполнено успешно.

```
x, err := strconv.ParseFloat("-99.7", 64)
fmt.Printf("%8T %6v %v\n", x, x, err)
y, err := strconv.ParseInt("71309", 10, 0)
fmt.Printf("%8T %6v %v\n", y, y, err)
z, err := strconv.Atoi("71309")
fmt.Printf("%8T %6v %v\n", z, z, err)
•float64•-99.7•<nil>
••int64•71309•<nil>
••••int•71309•<nil>
```

Функции strconv.ParseFloat(), strconv.ParseInt() и strconv.Atoi() («ASCII to int» – «ASCII в int»), показанные здесь, действуют практически так, как можно было бы ожидать. Вызов функции strconv.Atoi(s) почти идентичен вызову функции strconv.ParseInt(s, 10, 0), то есть он интерпретирует указанную строку как строковое представление целого числа в десятичной системе счисления и

возвращает целое число, только функция `Atoi()` возвращает значение типа `int`, а функция `ParseInt()` – значение типа `int64`. Как можно было бы догадаться, функция `strconv.ParseUint()` преобразует строку в беззнаковый целочисленный тип и терпит неудачу, если строка начинается со знака «минус». Все эти функции терпят неудачу, если в начале или в конце строки имеются пробельные символы, которые, впрочем, легко удалить с помощью функции `strings.TrimSpace()` или функций ввода из пакета `fmt` (табл. 8.2 ниже). Естественно, функции преобразования в вещественные значения принимают строки, содержащие числа в стандартном или экспоненциальном представлении, такие как `"984"`, `"424.019"` и `"3.916e-12"`.

```
s := strconv.FormatBool(z > 100)
fmt.Println(s)
i, err := strconv.ParseInt("0xDEED", 0, 32)
fmt.Println(i, err)
j, err := strconv.ParseInt("0707", 0, 32)
fmt.Println(j, err)
k, err := strconv.ParseInt("10111010001", 2, 32)
true
57069.<nil>
455.<nil>
1489.<nil>
```

Функция `strconv.FormatBool()` возвращает строковое представление указанного логического выражения в виде строки `"true"` или `"false"`. Функция `strconv.ParseInt()` преобразует строковое представление целого числа в значение типа `int64`. Во втором параметре функции передается основание системы счисления, где значение `0` означает, что основание системы счисления будет определено автоматически по первым символам в строке: `"0x"` или `"0X"` – шестнадцатеричная система счисления, `"0"` – восьмеричная, и во всех остальных случаях – десятичная. В этом фрагменте было выполнено преобразование шестнадцатеричного и восьмеричного чисел с автоматическим определением основания системы счисления, и двоичного числа с явно указанным значением `2` в аргументе `base`. Допустимыми значениями аргумента `base` являются значения в диапазоне от `2` до `36` включительно, где в основаниях выше `10` десятка представляется как `A` (или `a`), и т. д. Третий аргумент определяет размер результата в битах (`0` означает тип `int`), поэтому, несмотря на то что функция всегда возвращает значение типа `int64`, преобразование

будет успешным, только если полученное значение может быть преобразовано в целочисленный тип указанного размера.

```
i := 16769023
fmt.Println(strconv.Itoa(i))
fmt.Println(strconv.FormatInt(int64(i), 10))
fmt.Println(strconv.FormatInt(int64(i), 2))
fmt.Println(strconv.FormatInt(int64(i), 16))
16769023
16769023
11111111110111111111111111
ffdf
```

Функция `strconv.Itoa()` («Integer to ASCII» – «целое в ASCII») возвращает строковое представление аргумента типа `int` в системе счисления по основанию 10. Функция `strconv.FormatInt()` возвращает строковое представление аргумента типа `int64` в системе счисления с указанным основанием (которое обязательно должно быть указано и находиться в диапазоне от 2 до 36 включительно).

```
s = "Alle ønsker å være fri."
quoted := strconv.Quote(s)
fmt.Println(quoted)
fmt.Println(strconv.Unquote(quoted))
"Alle\u00f8nsker\u00e5v\u00e6re fri."
Alle\u00f8nsker\u00e5v\u00e6re fri.<nil>
```

Функция `strconv.Quote()` возвращает переданную ей строку в виде строки в кавычках, с учетом синтаксиса языка, где все непечатаемые символы ASCII, а также все символы, не являющиеся символами ASCII, представлены экранированными последовательностями. (Экранированные последовательности, поддерживаемые в языке Go, перечислены в табл. Table 3.1 выше.) Функция `strconv.Unquote()` принимает строку в кавычках или в обратных апострофах, или символ в апострофах, и возвращает эквивалентную строку без кавычек и значение ошибки (или `nil`).

3.6.3. Пакет *utf8*

Пакет `unicode/utf8` содержит несколько функций для выполнения операций со строками и срезами типа `[]bytes`, хранящими байты в

кодировке UTF-8, многие из которых показаны в табл. 3.10. Ранее уже демонстрировалось использование функций `utf8.DecodeRuneInString()` и `utf8.DecodeLastRuneInString()` (§3.3) для получения первого и последнего символов в строке.

Таблица 3.10. *Функции из пакета `utf8`*

Пакет импортируется под именем `"unicode/utf8"`. Переменная `b` – это срез типа `[]byte`, `s` – строка типа `string`, и `c` – кодовый пункт Юникода типа `rune`.

Функция	Описание/результат
<code>utf8.DecodeLastRune(b)</code>	Последний символ (значение типа <code>rune</code>) в <code>b</code> и количество байт, занимаемое им, или <code>U+FFFD</code> (символ замены «?»), и 0 – если <code>b</code> не заканчивается допустимым значением типа <code>rune</code>
<code>utf8.DecodeLastRuneInString(s)</code>	То же, что и функция <code>utf8.DecodeLastRune()</code> , только принимает строку, а не срез
<code>utf8.DecodeRune(b)</code>	Первый символ (значение типа <code>rune</code>) в <code>b</code> и количество байт, занимаемое им, или <code>U+FFFD</code> (символ замены «?»), и 0 – если <code>b</code> не начинается допустимым значением типа <code>rune</code>
<code>utf8.DecodeRuneInString(s)</code>	То же, что и функция <code>utf8.DecodeRune()</code> , только принимает строку, а не срез
<code>utf8.EncodeRune(b, c)</code>	Записывает <code>c</code> в <code>b</code> в виде последовательности байт в кодировке UTF-8 и возвращает число записанных байтов (в срезе <code>b</code> должно быть достаточно свободного места)
<code>utf8.FullRune(b)</code>	<code>true</code> , если <code>b</code> начинается с допустимого значения типа <code>rune</code> в кодировке UTF-8
<code>utf8.FullRuneInString(s)</code>	<code>true</code> , если <code>s</code> начинается с допустимого значения типа <code>rune</code> в кодировке UTF-8
<code>utf8.RuneCount(b)</code>	То же, что и функция <code>utf8.RuneCountInString()</code> , но работает со значением типа <code>[]byte</code>
<code>utf8.RuneCountInString(s)</code>	Количество символов типа <code>rune</code> в строке <code>s</code> ; это число может быть меньше, чем <code>len(s)</code> , если строка <code>s</code> содержит не ASCII-символы
<code>utf8.RuneLen(c)</code>	Число байтов, необходимых для кодирования символа <code>c</code>
<code>utf8.RuneStart(x)</code>	<code>true</code> , если байт <code>x</code> может быть первым байтом символа
<code>utf8.Valid(b)</code>	<code>true</code> , если байты в <code>b</code> представляют допустимые символы в кодировке UTF-8
<code>utf8.ValidString(s)</code>	<code>true</code> , если байты в <code>s</code> представляют допустимые символы в кодировке UTF-8

3.6.4. Пакет *unicode*

Пакет *unicode* содержит функции для получения кодовых пунктов Юникода и определения соответствия их некоторым критериям, например чтобы проверить, является символ цифрой или буквой нижнего регистра. В табл. 3.11 перечислены наиболее часто используемые функции. В дополнение к этим функциям в пакете также имеются такие функции, как `unicode.ToLower()` и `unicode.IsUpper()`, универсальная функция `unicode.Is()`, с помощью которой можно проверить принадлежность символа к той или иной категории Юникода.

Таблица 3.11. Функции из пакета *unicode*

Переменная *c* – это символ типа *rune* и представляет кодовый пункт Юникода.

Функция	Описание/результат
<code>unicode.Is(table, c)</code>	<code>true</code> , если символ <i>c</i> присутствует в таблице <i>table</i> (см. описание в тексте)
<code>unicode.IsControl(c)</code>	<code>true</code> , если символ <i>c</i> является управляющим символом
<code>unicode.IsDigit(c)</code>	<code>true</code> , если символ <i>c</i> является десятичной цифрой
<code>unicode.IsGraphic(c)</code>	<code>true</code> , если символ <i>c</i> является «графическим» символом, таким как буква, число, знак пунктуации, символ или пробел
<code>unicode.IsLetter(c)</code>	<code>true</code> , если символ <i>c</i> является буквой
<code>unicode.IsLower(c)</code>	<code>true</code> , если символ <i>c</i> является буквой нижнего регистра
<code>unicode.IsMark(c)</code>	<code>true</code> , если символ <i>c</i> является символом маркера
<code>unicode.IsOneOf(tables, c)</code>	<code>true</code> , если символ <i>c</i> присутствует в одной из таблиц <i>tables</i>
<code>unicode.IsPrint(c)</code>	<code>true</code> , если символ <i>c</i> является печатаемым символом
<code>unicode.IsPunct(c)</code>	<code>true</code> , если символ <i>c</i> является знаком пунктуации
<code>unicode.IsSpace(c)</code>	<code>true</code> , если символ <i>c</i> является пробельным символом
<code>unicode.IsSymbol(c)</code>	<code>true</code> , если символ <i>c</i> является символическим знаком
<code>unicode.IsTitle(c)</code>	<code>true</code> , если символ <i>c</i> является заглавной буквой
<code>unicode.IsUpper(c)</code>	<code>true</code> , если символ <i>c</i> является буквой верхнего регистра
<code>unicode.SimpleFold(c)</code>	Копия символа <i>c</i> в противоположном регистре
<code>unicode.To(case, c)</code>	Версия символа <i>c</i> в регистре <i>case</i> , где <i>case</i> может иметь значение <code>unicode.LowerCase</code> , <code>unicode.TitleCase</code> или <code>unicode.UpperCase</code>
<code>unicode.ToLower(c)</code>	Версия символа <i>c</i> в нижнем регистре
<code>unicode.ToTitle(c)</code>	Версия символа <i>c</i> в заглавном регистре
<code>unicode.ToUpper(c)</code>	Версия символа <i>c</i> в верхнем регистре

```
fmt.Println(IsHexDigit('8'), IsHexDigit('x'), IsHexDigit('X'),  
            IsHexDigit('b'), IsHexDigit('B'))  
true·false·false·true·true
```

В пакете `unicode` имеется функция `unicode.IsDigit()`, проверяющая, является ли символ десятичной цифрой, но в пакете нет аналогичной функции для проверки шестнадцатеричных цифр, поэтому здесь используется пользовательская функция `IsHexDigit()`.

```
func IsHexDigit(char rune) bool {  
    return unicode.Is(unicode.ASCII_Hex_Digit, char)  
}
```

В этой короткой функции для проверки, является ли указанный символ шестнадцатеричной цифрой, используется универсальная функция `unicode.Is()` в комбинации с диапазоном `unicode.ASCII_Hex_Digit`. Так же просто можно создать аналогичные функции для проверки других характеристик символов Юникода.

3.6.5. Пакет *regex*

Этот раздел содержит таблицы со списками функций из пакета `regex` и синтаксических конструкций регулярных выражений, поддерживаемых пакетом, а также несколько примеров. Далее в этом разделе и в оставшейся части книги будет предполагаться, что читатель уже имеет опыт работы с регулярными выражениями¹.

Пакет `regex` представляет собой реализацию механизма регулярных выражений RE2 Рассы Кокса (Russ Cox)². Это быстрый механизм, поддерживающий возможность выполнения в многопоточных программах. В механизме RE2 не выполняются *возвраты* (backtracking), что гарантирует линейное увеличение времени вы-

¹ Желая освоить регулярные выражения можно порекомендовать отличную книгу «Mastering Regular Expressions» – см. приложение С. В книге «Programming in Python 3», написанной автором, имеется глава о поддержке регулярных выражений (подмножества их синтаксиса) в языке Python. Эта глава доступна для свободной загрузки по адресу: www.informit.com/title/9780321680563 (щелкните на ссылке **Sample Content** (Пример содержимого) и загрузите главу 13).

² Информацию о механизме RE2, включая ссылки на документы, описывающие принцип действия, производительность и особенности реализации, можно найти по адресу: code.google.com/p/re2/.

полнения $O(n)$, где n – длина сопоставляемой строки, тогда как в механизмах с возвратами время выполнения может возрасти по экспоненте $O(n^2)$ (см. врезку «Нотация $O(\dots)$ », выше). Высокая производительность получена в основном за счет отказа от поддержки обратных ссылок. Однако это ограничение обычно легко можно обойти за счет применения функций из пакета `regexp`.

В табл. 3.12 перечислены функции из пакета `regexp`, включая четыре функции, создающие значения типа `*regexp.Regexp`. Методы этих значений перечислены в табл. 3.18 и 3.19 (ниже). В табл. 3.13 даны экранированные последовательности, поддерживаемые механизмом RE2, а в табл. 3.14 – классы символов. В табл. 3.15 приведены различные проверки, в табл. 3.16 – квантификаторы, в табл. 3.17 – флаги.

Оба метода, `regexp.Regexp.ReplaceAll()` и `regexp.Regexp.ReplaceAllString()`, поддерживают нумерованные и именованные ссылки. Нумерованные ссылки начинаются с `$1`, соответствующей первой сохраняющей паре круглых скобок. *Именованные ссылки* ссылаются на имена сохраняющих групп. Несмотря на то что ссылки могут указываться непосредственно по их номерам или именам (например, `$2`, `$filename`), надежнее будет заключать их в фигурные скобки (например, `${2}`, `${filename}`). Для включения литерала `$` в имя ссылки следует использовать пару символов `$$`.

Таблица 3.12. Функции из пакета `regexp`

Переменные `p` и `s` имеют тип `string`, где `p` – шаблон регулярного выражения.

Функция	Описание/результат
<code>regexp.Match(p, b)</code>	<code>true</code> и <code>nil</code> , если шаблону <code>p</code> соответствует <code>b</code> типа <code>[]byte</code>
<code>regexp.MatchReader(p, r)</code>	<code>true</code> и <code>nil</code> , если шаблону <code>p</code> соответствует текст, возвращаемый значением <code>r</code> типа <code>io.RuneReader</code>
<code>regexp.MatchString(p, s)</code>	<code>true</code> и <code>nil</code> , если строка <code>s</code> соответствует шаблону <code>p</code>
<code>regexp.QuoteMeta(s)</code>	Строка, в которой все метасимволы регулярных выражений надежно экранированы
<code>regexp.Compile(p)</code>	Значение <code>*regexp.Regexp</code> и <code>nil</code> в случае успешной компиляции шаблона <code>p</code> ; см. табл. 3.18 и 3.19 (ниже)
<code>regexp.CompilePOSIX(p)</code>	Значение <code>*regexp.Regexp</code> и <code>nil</code> в случае успешной компиляции шаблона <code>p</code> ; см. табл. 3.18 и 3.19 (ниже)
<code>regexp.MustCompile(p)</code>	Значение <code>*regexp.Regexp</code> в случае успешной компиляции шаблона <code>p</code> , иначе возникает аварийная ситуация; см. табл. 3.18 и 3.19 (ниже)
<code>regexp.MustCompilePOSIX(p)</code>	Значение <code>*regexp.Regexp</code> в случае успешной компиляции шаблона <code>p</code> , иначе возникает аварийная ситуация; см. табл. 3.18 и 3.19 (ниже)

Таблица 3.13. Экранированные последовательности, поддерживаемые пакетом *regex*

Последовательность	Описание
\с	Литерал символа с; например * – литерал символа *, а не квантификатор
\000	Символ с указанным восьмеричным кодовым пунктом
\хNN	Символ с указанным 2-значным шестнадцатеричным кодовым пунктом
\х{NNNN}	Символ с указанным 4-значным шестнадцатеричным кодовым пунктом
\a	ASCII-символ сигнала (BEL) ≡ \007
\f	ASCII-символ перевода формата (FF) ≡ \014
\n	ASCII-символ перевода строки (LF) ≡ \012
\r	ASCII-символ возврата каретки (CR) ≡ \015
\t	ASCII-символ горизонтальной табуляции (TAB) ≡ \011
\v	ASCII-символ вертикальной табуляции (VT) ≡ \013
\Q... \E	Соответствует тексту ... буквально, даже если он содержит такие символы, как *

Таблица 3.14. Классы символов, поддерживаемые пакетом *regex*

Последовательность	Описание
[символы]	Любой из перечисленных символов
[^символы]	Любой из неперечисленных символов
[:имя :]	Любой из ASCII-символов, входящий в класс с указанным именем: [[:alnum:]] ≡ [0-9A-Za-z] [[:lower:]] ≡ [a-z] [[:alpha:]] ≡ [A-Za-z] [[:print:]] ≡ [-~] [[:ascii:]] ≡ [\x00-\x7F] [[:punct:]] ≡ [!-/:-@[-' {-~] [[:blank:]] ≡ [\t] [[:space:]] ≡ [\t\n\v\f\r] [[:cntrl:]] ≡ [\x00-\x1F\x7F] [[:upper:]] ≡ [A-Z] [[:digit:]] ≡ [0-9] [[:word:]] ≡ [0-9A-Za-z_] [[:graph:]] ≡ [!-~] [[:xdigit:]] ≡ [0-9A-Fa-f]
[:^имя :]	Любой из ASCII-символов, не входящий в класс с указанным именем
.	Любой символ (включая перевод строки, если установлен флаг s)
\d	Любая ASCII-цифра: [0-9]
\D	Любой ASCII-символ, не являющийся цифрой: [^0-9]
\s	Любой пробельный ASCII-символ: [\t\n\f\r]
\S	Любой ASCII-символ, не являющийся пробельным: [^ \t\n\f\r]

Последовательность	Описание
\w	Любой ASCII-символ «слова»: [0-9A-Za-z]
\W	Любой ASCII-символ, не являющийся символом «слова»: [^0-9A-Za-z]
\pN	Любой символ Юникода, входящий в класс N с однобуквенным именем, например последовательности \pL соответствуют буквы Юникода
\PN	Любой символ Юникода, не входящий в класс N с однобуквенным именем, например последовательности \pL соответствуют символы Юникода, не являющиеся буквами
\p{Имя}	Любой символ Юникода, входящий в класс с именем Имя, например последовательности \p{Ll} соответствуют буквы нижнего регистра, последовательности \p{Lu} – буквы верхнего регистра, последовательности \p{Greek} – буквы греческого алфавита
\P{Имя}	Любой символ Юникода, не входящий в класс с именем Имя

Таблица 3.15. Проверки, поддерживаемые пакетом *regex*

Последовательность	Описание
^	Начало текста (или начало строки, если установлен флаг m)
\$	Конец текста (или конец строки, если установлен флаг m)
\A	Начало текста
\Z	Конец текста
\b	Граница слова (\w, за которым следует \W, или \A, или \Z, и наоборот)
\B	Не граница слова

Таблица 3.16. Квантификаторы, поддерживаемые пакетом *regex*

Квантификатор	Описание
e? или e{0, 1}	Соответствует нулю или одному совпадению максимальной длины с выражением e
e+ или e{1, }	Соответствует одному или более совпадениям максимальной длины с выражением e
e* или e{0, }	Соответствует нулю или более совпадениям максимальной длины с выражением e
e{m, }	Соответствует по меньшей мере m совпадениям максимальной длины с выражением e
e{, n}	Соответствует не более n совпадениям максимальной длины с выражением e

Таблица 3.16. Квантификаторы, поддерживаемые пакетом *regexpr*

$e\{m, n\}$	Соответствует от n до m совпадениям максимальной длины с выражением e
$e\{m\}$ или $e\{m\}?$	Соответствует точно m совпадениям с выражением e
$e??$ или $e\{0, 1\}?$	Соответствует нулю или одному совпадению минимальной длины с выражением e
$e+?$ или $e\{1, \}$?	Соответствует одному или более совпадениям минимальной длины с выражением e
$e*?$ или $e\{0, \}$?	Соответствует нулю или более совпадениям минимальной длины с выражением e
$e\{m, \}$?	Соответствует по меньшей мере m совпадениям минимальной длины с выражением e
$e\{, n\}?$	Соответствует не более n совпадениям минимальной длины с выражением e
$e\{m, n\}?$	Соответствует от n до m совпадениям минимальной длины с выражением e

Таблица 3.17. Флаги и группировки, поддерживаемые пакетом *regexpr*

Флаг или оператор группировки	Описание
<i>i</i>	Сопоставление выполняется без учета регистра символов (по умолчанию регистр символов учитывается)
<i>m</i>	Многострочный режим; якорные метасимволы \wedge и $\$$ соответствуют началу и концу каждой строки (по умолчанию используется однострочный режим)
<i>s</i>	Метасимволу $\.$ соответствуют любые символы, включая символы перевода строки (по умолчанию точке соответствуют любые символы, кроме символов перевода строки)
<i>U</i>	Превращает максимальные квантификаторы в минимальные и наоборот, то есть изменяет значение символа $?$, следующего за квантификатором (по умолчанию квантификаторы стараются отыскать совпадение максимальной длины, если за ними не следует символ $?$)
$(?флаги)$	Активирует указанные флаги с текущего момента (чтобы отключить флаг или флаги, перед ними следует поставить знак $-$)
$(?флаги:e)$	Применяет указанные флаги к выражению e (чтобы отключить флаг или флаги, перед ними следует поставить знак $-$)
(e)	Группирует и сохраняет совпадение с выражением e
$(?P<имя>e)$	Группирует и сохраняет совпадение с выражением e под указанным именем
$(?:e)$	Группирует, но не сохраняет совпадение с выражением e

Таблица 3.18. Методы типа **regexp.Regexp*, часть 1

Переменная *rx* имеет тип **regexp.Regexp*; *s* – строка, сопоставляемая с шаблоном; *b* – срез типа *[]byte*, сопоставляемый с шаблоном; *r* – значение, реализующее интерфейс *io.RuneReader*, сопоставляемое с шаблоном; *n* – максимальное число совпадений (–1 означает максимально возможное число совпадений).

Возвращаемое значение *nil* свидетельствует об отсутствии совпадений.

Метод	Описание/результат
<code>rx.Expand(...)</code>	Выполняет подстановку значений по ссылкам, как это делает метод <code>ReplaceAll()</code> ; редко используется непосредственно
<code>rx.ExpandString(...)</code>	Выполняет подстановку значений по ссылкам, как это делает метод <code>ReplaceAllString()</code> ; редко используется непосредственно
<code>rx.Find(b)</code>	Значение типа <i>[]byte</i> с самым первым найденным совпадением или <i>nil</i>
<code>rx.FindAll(b, n)</code>	Значение типа <i>[][]byte</i> со всеми неперекрывающимися совпадениями или <i>nil</i>
<code>rx.FindAllIndex(b, n)</code>	Значение типа <i>[][]int</i> (срез массива срезов, состоящих из двух элементов), где каждый элемент определяет совпадение или имеет значение <i>nil</i> , например <i>b[pos[0]:pos[1]]</i> , где <i>pos</i> – один из двух-элементных срезов
<code>rx.FindAllString(s, n)</code>	Значение типа <i>[]string</i> со всеми неперекрывающимися совпадениями или <i>nil</i>
<code>rx.FindAllStringIndex(s, n)</code>	Значение типа <i>[][]int</i> (срез массива срезов, состоящих из двух элементов), где каждый элемент определяет совпадение или имеет значение <i>nil</i> , например <i>s[pos[0]:pos[1]]</i> , где <i>pos</i> – один из двух-элементных срезов
<code>rx.FindAllStringSubmatch(s, n)</code>	Значение типа <i>[][]string</i> (срез массива срезов строк, где каждая строка соответствует значению сохраняющей группы) или <i>nil</i>
<code>rx.FindAllStringSubmatchIndex(s, n)</code>	Значение типа <i>[][]int</i> (срез массива срезов, состоящих из двух элементов типа <i>int</i> , соответствующих значениям сохраняющих групп) или <i>nil</i>
<code>rx.FindAllSubmatch(b, n)</code>	Значение типа <i>[][][]byte</i> (срез массива срезов типа <i>[]byte</i> , где каждое значение <i>[]byte</i> соответствует значению сохраняющей группы) или <i>nil</i>
<code>rx.FindAllSubmatchIndex(b, n)</code>	Значение типа <i>[][]int</i> (срез массива срезов, состоящих из двух элементов типа <i>int</i> , соответствующих значениям сохраняющих групп) или <i>nil</i>
<code>rx.FindIndex(b)</code>	Двухэлементный срез типа <i>[]int</i> , определяющий самое первое совпадение, например <i>b[pos[0]:pos[1]]</i> , где <i>pos</i> – один из двухэлементных срезов, или <i>nil</i>

Таблица 3.18. Методы типа **regexp.Regexp*, часть 1

Метод	Описание/результат
<code>rx.FindReader-Index(r)</code>	Двухэлементный срез типа <code>[]int</code> , определяющий самое первое совпадение и значение сохраняющей группы, или <code>nil</code>
<code>rx.FindReader-SubmatchIndex(r)</code>	Значение типа <code>[]int</code> , определяющее самое первое совпадение и значение сохраняющей группы, или <code>nil</code>
<code>rx.FindString(s)</code>	Самое первое совпадение или пустая строка
<code>rx.FindString-Index(s)</code>	Двухэлементный срез типа <code>[]int</code> , определяющий самое первое совпадение, или <code>nil</code>
<code>rx.FindString-Submatch(s)</code>	Значение типа <code>[]string</code> с самым первым совпадением и значением сохраняющей группы или <code>nil</code>
<code>rx.FindString-SubmatchIndex(s)</code>	Значение типа <code>[]int</code> , определяющее самое первое совпадение и значение сохраняющей группы, или <code>nil</code>

Таблица 3.19. Методы типа **regexp.Regexp*, часть 2

Переменная `rx` имеет тип `*regexp.Regexp`; `s` – строка, сопоставляемая с шаблоном; `b` – срез типа `[]byte`, сопоставляемый с шаблоном.

Метод	Описание/результат
<code>rx.FindSubmatch(b)</code>	Значение типа <code>[]byte</code> с самым первым совпадением и значением сохраняющей группы или <code>nil</code>
<code>rx.FindSubmatch-Index(b)</code>	Значение типа <code>[]int</code> с самым первым совпадением и значением сохраняющей группы или <code>nil</code>
<code>rx.LiteralPrefix()</code>	Возможно, пустая строка префикса, с которой должно начинаться любое совпадение с регулярным выражением, и значение типа <code>bool</code> , указывающее, соответствует ли регулярному выражению вся строка целиком
<code>rx.Match(b)</code>	<code>true</code> , если <code>b</code> соответствует регулярному выражению
<code>rx.MatchReader(r)</code>	<code>true</code> , если содержимое значения <code>r</code> типа <code>io.RuneReader</code> соответствует регулярному выражению.
<code>rx.MatchString(s)</code>	<code>true</code> , если <code>s</code> соответствует регулярному выражению
<code>rx.NumSubexp()</code>	Количество групп в круглых скобках, содержащихся в регулярном выражении
<code>rx.ReplaceAll(b, br)</code>	Значение типа <code>[]byte</code> – копия аргумента <code>b</code> , где каждое совпадение замещено значениями ссылок, имена которых указаны в срезе <code>br</code> типа <code>[]byte</code> (см. описание в тексте)
<code>rx.ReplaceAll-Func(b, f)</code>	Значение типа <code>[]byte</code> – копия аргумента <code>b</code> , где каждое совпадение замещено возвращаемым значением, полученным от функции <code>f</code> с сигнатурой <code>func([]byte) []byte</code> , которой передаются найденные совпадения

Таблица 3.19. Методы типа **regexp.Regexp*, часть 2

<code>rx.ReplaceAllLiteral(b, br)</code>	Значение типа <code>[]byte</code> – копия аргумента <code>b</code> , где каждое совпадение замещено значением <code>br</code> типа <code>[]byte</code> (см. описание в тексте)
<code>rx.ReplaceAllLiteralString(s, sr)</code>	Значение типа <code>string</code> – копия аргумента <code>s</code> , где каждое совпадение замещено значением <code>sr</code> (см. описание в тексте)
<code>rx.ReplaceAllString(s, sr)</code>	Значение типа <code>string</code> – копия аргумента <code>s</code> , где каждое совпадение замещено значениями ссылок, имена которых указаны в строке <code>sr</code> (см. описание в тексте)
<code>rx.ReplaceAllStringFunc(s, f)</code>	Значение типа <code>string</code> – копия аргумента <code>s</code> , где каждое совпадение замещено возвращаемым значением, полученным от функции <code>f</code> с сигнатурой <code>func(string) string</code> , которой передаются найденные совпадения
<code>rx.String()</code>	Значение типа <code>string</code> , содержащее текст регулярного выражения
<code>rx.Subexp-Names()</code>	Значение типа <code>[]string</code> (которое не должно изменяться), содержащее имена всех именованных подвыражений

Типичным примером использования ссылок является операция преобразования списка имен, имеющего вид: *имя1...имяN фамилия*, в список, имеющий вид: *фамилия, имя1...имяN*. Ниже показано, как это можно реализовать с помощью пакета `regexp`, обеспечив корректную обработку символов с диакритическими знаками и других национальных символов.

```
nameRx := regexp.MustCompile(`(\pL\.(?:\s\pL\.)*)\s+(\pL+)`)
for i := 0; i < len(names); i++ {
    names[i] = nameRx.ReplaceAllString(names[i], "${2}, ${1}")
}
```

Переменная `names` имеет тип `[]string` и изначально хранит оригинальный список имен. По завершении цикла переменная `names` будет хранить измененный список имен.

Регулярному выражению соответствует одно или более имен, разделенных пробельными символами, каждое из которых состоит из одной или более букв Юникода (`\pL`) с последующей необязательной запятой и фамилией из одной или более букв Юникода.

Использование *нумерованных ссылок* может привести к проблемам в процессе сопровождения, например если в середину вставить

еще одну сохраняющую группу, тогда одна из ссылок окажется ошибочной. Решить эту проблему можно использованием *именованных ссылок*, которые не зависят от порядка следования соответствующих им именованных групп.

```
nameRx := regexp.MustCompile(
    `(?P<forenames>\pL+\.(?:\s+\pL+\.?)*)\s+(?P<surname>\pL+)`)
for i := 0; i < len(names); i++ {
    names[i] = nameRx.ReplaceAllString(names[i],
        "${surname}, ${forenames}")
}
```

Здесь определены две сохраняющие группы с осмысленными именами. Это помогло сделать более очевидными регулярное выражение и строку замены со ссылками.

Простейшее выражение поиска дубликатов «слов», опирающееся на обратные ссылки, на языке Python или Perl можно было бы записать так: `\b(\w+)\s+\1\b`. Поскольку пакет `regexp` не поддерживает обратных ссылок, для достижения того же эффекта необходимо объединить регулярное выражение с несколькими строками программного кода.

```
wordRx := regexp.MustCompile(`\w+`)
if matches := wordRx.FindAllString(text, -1); matches != nil {
    previous := ""
    for _, match := range matches {
        if match == previous {
            fmt.Println("Duplicate word:", match)
        }
        previous = match
    }
}
```

Регулярному выражению соответствует один или более символов «слова». Функция `regexp.Regexp.FindAllString()` возвращает значение типа `[]string`, содержащее все неперекрывающиеся совпадения. Если в испытуемой строке будет найдено хотя бы одно совпадение (то есть переменная `matches` будет иметь значение, отличное от `nil`), фрагмент выполнит итерации по срезу со строками и выведет все дубликаты, сравнивая каждое найденное слово с предыдущим.

Другой типичный пример использования регулярных выражений: извлечение пар *ключ:значение* из файлов с настройками. Ниже

приводится фрагмент, заполняющий отображение информацией, полученной из таких строк.

```
valueForKey := make(map[string]string)
keyValueRx := regexp.MustCompile(`\s*([[:alpha:]]\w*)\s*:\s*(.+)`)
if matches := keyValueRx.FindAllStringSubmatch(lines, -1); matches != nil {
    for _, match := range matches {
        valueForKey[match[1]] = strings.TrimRight(match[2], "\t ")
    }
}
```

Регулярное выражение пропускает любые пробельные символы в начале строки и совпадает с ключом, обязательно начинающимся с буквы латинского алфавита, за которой может следовать нуль или более букв, цифр или символов подчеркивания, необязательный пробельный символ, двоеточие, необязательный пробельный символ, и значение – любые символы до конца строки или до символа перевода строки, не включая его. В данном случае вместо символьного класса `[[:alpha:]]` можно было бы использовать нечто более краткую форму записи `[A-Za-z]`, а для поддержки ключей, содержащих любые символы Юникода, – подвыражение `(\pL[\pL\p{Nd}_]*)` – буква Юникода, за которой следует нуль или более букв Юникода, десятичных цифр или символов подчеркивания. Поскольку подвыражение `.+` не совпадает с символом перевода строки, данное регулярное выражение может применяться к тексту, содержащему множество строк с парами *ключ:значение*.

Благодаря использованию максимального квантификатора регулярное выражение поглотит все пробельные символы, предшествующие значению. Но, чтобы избавиться от ненужных пробельных символов в конце значения, придется воспользоваться функцией усечения строки, потому что добавление подвыражения `\s*` в конец не даст никакого эффекта из-за максимального квантификатора в подвыражении `.+`. При этом здесь нельзя использовать минимальный квантификатор (такой как `.+?`), потому что в этом случае он совпадет только с первым словом в значении, содержащем два или более слова, разделенных пробелами.

Функция `regexp.Regexp.FindAllStringSubmatch()` возвращает срез срезов со строками (или `nil`). Значение `-1` во втором аргументе сообщает функции, что она должна отыскать максимально возможное количество (неперекрывающихся) совпадений. В данном примере

для каждого совпадения создается срез точно с тремя строками, где первая содержит полное совпадение, вторая – ключ, третья – значение. Обе строки, ключ и значение, будут содержать как минимум по одному символу, потому что таковы минимальные требования к совпадению.

Обычно для анализа разметки XML предпочтительнее использовать значение `xml.Decoder`, однако иногда элементы XML-документа содержат простые пары атрибутов, имеющие вид: *имя*="значение" или *имя*'значение', для извлечения которых достаточно простого регулярного выражения.

```
attrValueRx := regexp.MustCompile(regexp.QuoteMeta(attrName) +
    `=(?:"([^\"]+)"|'([^']+)'`)`)
if indexes := attrValueRx.FindAllStringSubmatchIndex(attrs, -1);
    indexes != nil {
    for _, positions := range indexes {
        start, end := positions[2], positions[3]
        if start == -1 {
            start, end = positions[4], positions[5]
        }
        fmt.Printf("%s'\n", attrs[start:end])
    }
}
```

Регулярному выражению `attrValueRx` соответствует надежно экранированное имя атрибута, за которым следуют знак равенства и строка в кавычках или апострофах. Круглые скобки, используемые для реализации выбора (`|`), также можно было бы использовать для сохранения найденного совпадения, но в данном случае это нежелательно – здесь не требуется сохранять значение с кавычками или апострофами, поэтому группирующие скобки сделаны несохраняющими (`(?:)`). Только чтобы показать, как это делается, вместо фактических строк совпадений здесь извлекаются их индексы. В данном примере всегда будут возвращаться три пары индексов (`[start:end]`), первая пара соответствует всему совпадению целиком, вторая пара – значению в кавычках, и третья пара – значению в апострофах. Разумеется, в каждом конкретном случае существовать будет только какая-то одна пара, а индексы в другой паре будут иметь значение `-1`.

Как и в предыдущих примерах, здесь извлекаются все неперекрывающиеся совпадения, найденные в строке, в данном случае – в виде

индексов `[[int (или nil)`]. В каждом срезе с индексами всему совпадению соответствует срез `attrs[positions[0]:positions[1]]`. А строке значения – срез `attrs[positions[2]:positions[3]]` или `attrs[positions[4]:positions[5]]`, в зависимости от типа кавычек, в которые заключено это значение. Код, выполняемый в цикле, начинается с предположения, что значение заключено в кавычки, но если это не так (то есть если `start == -1`), тогда используется срез со значением в апострофах.

Выше (§3.6.1) демонстрировалось, как можно реализовать функцию `SimplifyWhitespace()`. Здесь показано, как можно добиться того же эффекта с помощью регулярного выражения и функции `strings.TrimSpace()`.

```
simplifyWhitespaceRx := regexp.MustCompile(`[\s\p{Zl}\p{Zp}]+`)
text = strings.TrimSpace(simplifyWhitespaceRx.ReplaceAllLiteralString(
    text, " "))
```

Регулярное выражение выполняет единственный проход по строке, а функция `strings.TrimSpace()` вызывается только в конце, поэтому данная комбинация выполняет не так много работы. Функция `regexp.Regexp.ReplaceAllLiteralString()` принимает исходную строку и текст замены, которым замещает все найденные совпадения. (Разница между `regexp.Regexp.ReplaceAllString()` и `regexp.Regexp.ReplaceAllLiteralString()` состоит в том, что первая получает замещающий текст по именам ссылок, а последняя – нет.) То есть в данном случае каждая последовательность из одного или более пробельных символов (пробельные ASCII-символы, а также символы Юникода – разделители строк и абзацев) будет замещена единственным пробелом.

В заключительном примере использования регулярных выражений демонстрируется, как реализовать замену с помощью функции.

```
unaccentedLatin1Rx := regexp.MustCompile(
    `[ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝàáâãäåæçèéêëìíîïðñòóôõöøùúûüýÿ]+`)
unaccented := unaccentedLatin1Rx.ReplaceAllStringFunc(latin1,
    UnaccentedLatin1)
```

Регулярному выражению здесь соответствует одна или более букв с диакритическим знаком из набора символов Latin-1. Функция `regexp.Regexp.ReplaceAllStringFunc()` вызывает функцию (с сигнатурой `func(string) string`), переданную ей во втором аргументе, для каждого найденного совпадения. Функции передается текст совпадения

в виде аргумента и замещается возвращаемым ею значением (которое может быть пустой строкой).

```
func UnaccentedLatin1(s string) string {
    chars := make([]rune, 0, len(s))
    for _, char := range s {
        switch char {
            case 'À', 'Á', 'Â', 'Ã', 'Ä', 'Å':
                char = 'A'
            case 'Æ':
                chars = append(chars, 'A')
                char = 'E'
            // ...
            case 'Ý', 'ÿ':
                char = 'y'
        }
        chars = append(chars, char)
    }
    return string(chars)
}
```

Эта простая функция обеспечивает замену всех букв с диакритическими знаками из набора Latin-1 их аналогами без диакритических знаков. Она также обеспечивает замену лигатуры *æ* (которая в некоторых языках играет роль полноправного символа) символами *a* и *e*. Конечно, это достаточно искусственный пример, поскольку здесь преобразование можно было бы выполнить простой инструкцией `unaccented := UnaccentedLatin1(latin1)`.

На этом завершается демонстрация примеров использования регулярных выражений. Обратите внимание, что в табл. 3.18 и 3.19 каждой функции, содержащей в имени слово «String», соответствует функция с именем без слова «String», которая оперирует не строкой, а значением типа `[]byte`. Кроме того, в книге имеются еще несколько примеров использования регулярных выражений (например, в §1.6 и §7.2.4.1).

После рассказа о строках и пакетах, предназначенных для обработки строк, глава завершается примером использования некоторых строковых функций, за которыми, как обычно, следуют несколько упражнений.

3.7. Пример: m3u2pls

В этом разделе будет рассмотрена небольшая, но законченная программа, которая читает произвольные файлы `.m3u` со списками

произведений и создает эквивалентные им файлы .pls. В программе широко используется пакет `strings` и применяются другие знания, полученные в этой и предыдущих главах, а также демонстрируются несколько новинок.

Ниже приводится фрагмент файла .m3u, где многоточием (...) обозначается большая часть списка произведений.

```
#EXTM3U
#EXTINF:315,David Bowie - Space Oddity
Music/David Bowie/Singles 1/01-Space Oddity.ogg
#EXTINF:-1,David Bowie - Changes
Music/David Bowie/Singles 1/02-Changes.ogg
...
#EXTINF:251,David Bowie - Day In Day Out
Music/David Bowie/Singles 2/18-Day In Day Out.ogg
```

Файл начинается со строкового литерала `#EXTM3U`. Каждое произведение в списке представлено двумя строками. Первая строка начинается со строкового литерала `#EXTINF:`, за которым следуют продолжительность произведения в секундах, запятая и название произведения. Продолжительность, равная `-1`, означает, что длина произведения неизвестна (в обоих форматах). Вторая строка – это путь к файлу с произведением. Здесь перечислены файлы в открытом формате *Vorbis Audio*, в контейнере *Ogg* (www.vorbis.com), и используются разделители элементов пути в стиле ОС Unix.

Ниже приводится фрагмент эквивалентного файла .pls. Здесь снова многоточие обозначает большую часть списка произведений.

```
[playlist]
File1=Music/David Bowie/Singles 1/01-Space Oddity.ogg
Title1=David Bowie - Space Oddity
Length1=315
File2=Music/David Bowie/Singles 1/02-Changes.ogg
Title2=David Bowie - Changes
Length2=-1
...
File33=Music/David Bowie/Singles 2/18-Day In Day Out.ogg
Title33=David Bowie - Day In Day Out
Length33=251
NumberOfEntries=33
Version=2
```

Формат файлов `.pls` немного сложнее формата `.m3u`. Файл начинается со строкового литерала `[playlist]`. Каждое произведение представлено тремя парами *ключ/значение*, определяющими имя файла, название и продолжительность в секундах. Фактически формат `.pls` является специализированной версией файлов `.ini` (файлы инициализации в Windows), где каждый ключ (название каждого раздела в квадратных скобках) должен быть уникальным — этим объясняется использование порядковых номеров в именах ключей. И завершается файл двумя строками с метаданными.

Программа `m3u2pls` (в файле `m3u2pls/m3u2pls.go`) ожидает получить имя файла с расширением `.m3u` в виде аргумента командной строки и на его основе создает файл с расширением `.pls`, выводя его в поток `os.Stdout` (то есть в консоль). Сохранить результаты работы в настоящем дисковом файле можно с помощью операции перенаправления. Ниже приводится пример использования программы.

```
$ ./m3u2pls Bowie-Singles.m3u > Bowie-Singles.pls
```

Здесь программе передается файл `Bowie-Singles.m3u`, и с помощью операции перенаправления результаты преобразования сохраняются в файле `Bowie-Singles.pls`, в формате `.pls`. (Было бы неплохо иметь возможность и обратного преобразования, поэтому реализация данной возможности будет предложена в качестве самостоятельного упражнения в следующем разделе.)

Программа будет представлена в этом разделе почти полностью, исключение составляет только раздел импорта пакетов.

```
func main() {
    if len(os.Args) == 1 || !strings.HasSuffix(os.Args[1], ".m3u") {
        fmt.Printf("usage: %s <file.m3u>\n", filepath.Base(os.Args[0]))
        os.Exit(1)
    }

    if rawBytes, err := ioutil.ReadFile(os.Args[1]); err != nil {
        log.Fatal(err)
    } else {
        songs := readM3uPlaylist(string(rawBytes))
        writePlsPlaylist(songs)
    }
}
```

Функция `main()` начинается с проверки наличия аргумента командной строки, содержащего имя файла с расширением `.m3u`. Функция `strings.HasSuffix()` принимает две строки и возвращает `true`, если первая строка оканчивается второй. Если имя файла с расширением `.m3u` не было указано, выводится сообщение о порядке использования, и программа завершается. Функция `filepath.Base()` возвращает базовое имя (то есть имя файла), извлекая его из строки полного пути к файлу, а функция `os.Exit()` корректно завершает работу программы – останавливает все go-подпрограммы и закрывает все открытые файлы – и возвращает свой аргумент операционной системе.

Если имя файла с расширением `.m3u` было указано пользователем, предпринимается попытка прочитать файл целиком с помощью функции `ioutil.ReadFile()`. Эта функция возвращает все содержимое файла (в виде значения типа `[]byte`) и значение ошибки, которое будет равно `nil`, если при чтении файла не возникло никаких проблем. В противном случае (например, если файл с указанным именем не существует или недоступен для чтения), вызовом функции `log.Fatal()` в консоль (в действительности – в поток `os.Stderr`) выводится сообщение об ошибке и программа завершается с кодом 1.

Если операция чтения увенчалась успехом, содержимое файла из двоичного представления преобразуется в строку (под двоичным представлением здесь подразумевается текст, содержащий 7-битные символы ASCII или символы Юникода в кодировке UTF-8), которая немедленно передается функции `readM3uPlaylist()` для дальнейшей обработки. Эта функция возвращает срез со значениями типа `Songs` (то есть `[]Song`). Затем полученные данные выводятся с помощью функции `writePlsPlaylist()`.

```
type Song struct {  
    Title string  
    Filename string  
    Seconds int  
}
```

Здесь определяется составной тип `Song struct` (§6.4), чтобы обеспечить удобный способ представления информации о каждом произведении, не зависящий от формата файла.

```
func readM3uPlaylist(data string) (songs []Song) {  
    var song Song  
    for _, line := range strings.Split(data, "\n") {
```

```
line = strings.TrimSpace(line)
if line == "" || strings.HasPrefix(line, "#EXTM3U") {
    continue
}
if strings.HasPrefix(line, "#EXTINF:") {
    song.Title, song.Seconds = parseExtinfLine(line)
} else {
    song.Filename = strings.Map(mapPlatformDirSeparator, line)
}
if song.Filename != "" && song.Title != "" && song.Seconds != 0 {
    songs = append(songs, song)
    song = Song{}
}
}
return songs
}
```

Эта функция принимает содержимое файла `.m3u` целиком в виде единственной строки и возвращает срез, содержащий все произведения, которые удалось извлечь из строки. Она начинается с объявления пустой переменной `song` типа `Song`. Благодаря принятой в языке Go практике всегда инициализировать переменные нулевыми значениями, первоначально переменная `song` содержит две пустые строки и значение `0` в поле `Song.Seconds`.

Сердцем функции является цикл `for ...range (§5.3)`. Здесь с помощью функции `strings.Split()` строка с содержимым файла `.m3u` разбивается на отдельные строки, по которым выполняет итерации цикл `for`. Если очередная строка оказывается пустой или если оказывается первой строкой в файле (то есть начинающейся со строкового литерала `"#EXTM3U"`), выполняется инструкция `continue` – она просто передает управление обратно в начало цикла `for` и запускает следующую итерацию, или в конец цикла, если не осталось строк для выполнения итераций.

Если строка начинается со строкового литерала `"#EXTINF:"`, она передается функции `parseExtinfLine()` для анализа: эта функция возвращает значения типа `string` и `int`, которые немедленно присваиваются полям `Song.Title` и `Song.Seconds` переменной `song`. В противном случае предполагается, что строка содержит имя файла (включая путь) к текущему произведению.

Однако строка с именем файла сохраняется не в том виде, в каком она хранилась в файле. Вместо этого имя файла

передается функции `strings.Map()` вместе со ссылкой на функцию `mapPlatformDirSeparator()`, преобразующей символы-разделители элементов пути в символы, используемые в качестве разделителей на текущей платформе, а уже полученная в результате строка сохраняется в поле `Song.Filename` переменной `song`. Функции `strings.Map()` передается ссылка на функцию отображения с сигнатурой `func(rune) rune` и значение типа `string`. Она вызывает функцию отображения для каждого символа в строке и замещает символ в исходной строке символом, полученным от функции отображения, который, разумеется, может быть тем же самым символом, что и оригинальный. Как это принято в языке Go, символ имеет тип `rune`, а его значением является кодовый пункт Юникода.

Если имя файла с произведением и название не являются пустыми строками, а также продолжительность не равна нулю, текущее значение переменной `song` добавляется в конец среза `songs` (типа `[] Song`), после чего ей присваивается пустая структура `Song` (две пустые строки и 0).

```
func parseExtinfLine(line string) (title string, seconds int) {
    if i := strings.IndexAny(line, "-0123456789"); i > -1 {
        const separator = ","
        line = line[i:]
        if j := strings.Index(line, separator); j > -1 {
            title = line[j+len(separator):]
            var err error
            if seconds, err = strconv.Atoi(line[j:]); err != nil {
                log.Printf("failed to read the duration for '%s': %v\n",
                    title, err)
                seconds = -1
            }
        }
    }
    return title, seconds
}
```

Эта функция используется для анализа строк вида: `#EXTINF:продолжительность,название`, где ожидается, что продолжительность будет представлена целым числом больше нуля или `-1`.

Функция `strings.IndexAny()` используется для определения позиции первой цифры или знака «минус». Значение позиции, равное `-1`, означает, что поиск не увенчался успехом; любое другое значение

является номером позиции первого вхождения любого из символов в строке, переданной функции `strings.IndexAny()` во втором аргументе, – в этом случае в переменной `i` оказывается номер позиции первой цифры значения продолжительности (или символа «–»).

После определения позиции первой цифры от строки отсекается начальная ее часть до этой позиции, то есть текст `"#EXTINF:"`, находящийся в начале строки, поэтому теперь строка приобретает вид: продолжительность, название.

Вызов функции `strings.Index()` во второй инструкции `if` возвращает позицию первого вхождения строки `","` в исследуемой строке или `-1`, если искомая строка не будет обнаружена.

Название – это текст от запятой и до конца строки. Чтобы извлечь текст, расположенный за запятой, необходимо к начальной позиции запятой (`j`) добавить количество байтов, занимаемых запятой (`len(separator)`). Конечно, здесь известно, что запятая является 7-битным ASCII-символом и занимает один байт, но тут представлен подход, пригодный для обработки любых символов Юникода, независящий от количества байтов, используемых для их представления.

Продолжительность – это число, занимающее часть строки с первой позиции и до позиции `j` (где находится запятая), исключая ее. Преобразование числа в значение типа `int` выполняется с помощью функции `strconv.Atoi()`. Если преобразование потерпело неудачу, продолжительность просто устанавливается равной значению `-1`, обозначающему «неизвестную продолжительность», и выводится соответствующее сообщение, чтобы известить пользователя о проблеме.

```
func mapPlatformDirSeparator(char rune) rune {
    if char == '/' || char == '\\' {
        return filepath.Separator
    }
    return char
}
```

Эта функция вызывается функцией `strings.Map()` (внутри функции `readM3uPlaylist()`) для каждого символа в имени файла. Она замещает разделители элементов пути в имени файла символами-разделителями, используемыми на текущей платформе. Все остальные символы возвращаются без изменений.

Подобно большинству кросс-платформенных языков программирования и библиотек, в языке Go для внутренних операций

используются символы-разделители в стиле ОС Unix на всех платформах, даже в Windows. Однако при подготовке информации для представления пользователю или для записи в файлы предпочтительнее использовать разделители, используемые на текущей платформе. Для этого можно воспользоваться константой `filepath.Separator`, которая в Unix-подобных системах хранит символ `/`, а в Windows – символ `\`.

В этом примере заранее неизвестно, какие символы-разделители будут использоваться в путях к файлам в списке произведений, прямые или обратные слешы, поэтому необходимо предусмотреть обработку обоих символов. Однако, если бы было известно наверняка, что в путях к файлам используется символ прямого слеша, можно было бы воспользоваться функцией `filepath.FromSlash()`: в Unix-подобных системах она возвращает пути к файлам без изменений и замещает символы прямого слеша символами обратного слеша в Windows.

```
func writePlsPlaylist(songs []Song) {
    fmt.Println("[playlist]")
    for i, song := range songs {
        i++
        fmt.Printf("File%d=%s\n", i, song.Filename)
        fmt.Printf("Title%d=%s\n", i, song.Title)
        fmt.Printf("Length%d=%d\n", i, song.Seconds)
    }
    fmt.Printf("NumberOfEntries=%d\nVersion=2\n", len(songs))
}
```

Эта функция выводит данные из `songs` в формате `.pls`. Данные выводятся в поток `os.Stdout` (то есть в консоль), поэтому для вывода данных в файл следует использовать операцию перенаправления.

Функция начинается с вывода заголовка раздела ("`[playlist]`") и затем для каждого произведения выводит имя файла, название и продолжительность в секундах в отдельных строках. Поскольку каждый ключ должен быть уникальным, к именам ключей добавляются порядковые номера, начиная с 1. В конце выводятся две строки с метаданными.

3.8. Упражнения

В этой главе предлагается выполнить два упражнения. Первое связано с изменением уже имеющейся программы командной строки. Для выполнения второго упражнения необходимо создать веб-приложение (необязательно).

1. Программа `m3u2pls` из предыдущего раздела прекрасно справляется с преобразованием файлов `.m3u` со списком произведений в формат `.pls`. Но ее можно сделать еще полезнее, если добавить в нее возможность обратного преобразования, из формата `.pls` в формат `.m3u`. Для выполнения этого упражнения скопируйте содержимое каталога `m3u2pls` в каталог `my_playlist`, например, и создайте новую программу с именем `playlist`, обладающую необходимой функциональностью. Она должна выводить следующее сообщение о порядке использования: `playlist <file.[pls|m3u]>`.

Если при запуске программе передается файл `.m3u`, она должна делать то же самое, что делает программа `m3u2pls`: выводить в консоль данные из файла в формате `.pls`. Но если программе передается файл `.pls`, она должна выводить в консоль данные в формате `.m3u`. Реализация новой функциональности займет примерно 50 строк кода. Простейшее решение представлено в файле `playlist/playlist.go`.

2. Приложения выборки, сопоставления и обработки данных, содержащих имена людей, часто позволяют получить более оптимальные результаты, сопоставляя имена по их произношению, а не по написанию. Существует множество алгоритмов сопоставления имен на английском языке, но самым старым и самым простым является алгоритм Soundex (созвучие).

Классический алгоритм Soundex производит индекс созвучия, состоящий из заглавной буквы, за которой следуют три цифры. Например, большинство алгоритмов Soundex производят один и тот же индекс созвучия «R163» для обоих имен, «Robert» и «Rupert». Однако для имен «Ashcroft» и «Ashcraft» некоторые алгоритмы Soundex (включая и тот, что используется в примере решения данного упражнения) производят индекс созвучия «A226», а другие – индекс «A261».

В данном упражнении предлагается написать веб-приложение, состоящее из двух веб-страниц. Первая страница (имеющая путь `/`) должна предлагать для заполнения простую форму, где пользователь мог бы ввести одно или более имен и получить их индексы созвучия, – она показана на рис. 3.3 слева. Вторая страница (имеющая путь `/test`) должна применять функцию `soundex()`, имеющуюся в приложении, к списку строк и сравнивать каждый результат с ожидаемым значением – она показана на рис. 3.3 справа.

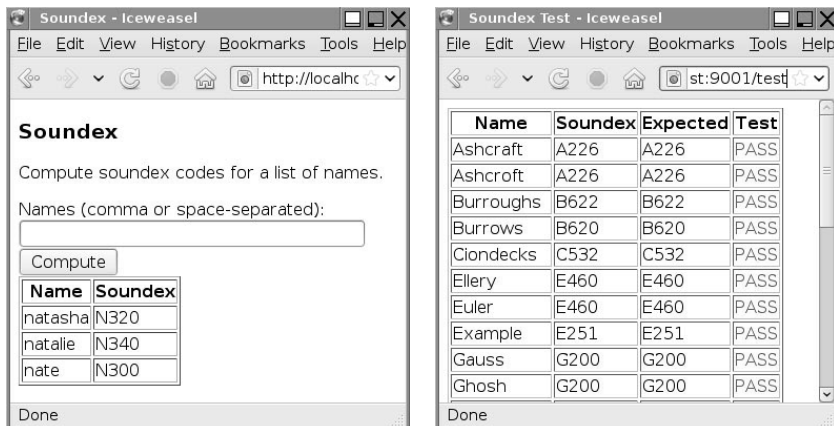


Рис. 3.3. Приложение Soundex в Linux

Читатели, предпочитающие сразу же приступить к исследованиям, минуя начальный этап, могут взять в качестве заготовки одно из имеющихся веб-приложений (`statistics`, `statistics_ans`, `quadratic_ans1`, `quadratic_ans2`) и сразу же сосредоточиться на реализации алгоритма вычисления индекса созвучия и тестовой страницы.

Пример решения, находящегося в файле `soundex/soundex.go`, содержит около 150 строк; сама функция `soundex()` занимает 20 строк, хотя она основана на использовании значения типа `[]int` для отображения заглавных букв в цифры немного необычным способом. Реализация решения основана на реализации алгоритма на языке Python, которая демонстрируется на веб-сайте Rosetta Code (rosettacode.org/wiki/Soundex). Однако реализация на языке Go, реализация, показанная на веб-сайте Rosetta Code и в Википедии (en.wikipedia.org/wiki/Soundex¹), производят немного отличающиеся результаты. Тестовые данные можно найти в файле `soundex/soundex-test-data.txt`.

Естественно, читатели могут реализовать любую понравившуюся версию алгоритма или даже взять за основу более сложный алгоритм, такой как алгоритм Metaphone, и просто скорректировать испытательные тесты.

¹ В Википедии имеется аналогичная страница, где рассматривается возможность определения индекса созвучия для русского языка: ru.wikipedia.org/wiki/Soundex. – Прим. перев.



4. Типы коллекций

В первом разделе этой главы описываются значения, указатели и ссылочные типы, так как эти знания потребуются в оставшейся части этой главы и в последующих главах. *Указатели* в языке Go действуют подобно указателям в языках C и C++ как синтаксически, так и семантически. Исключение составляет отсутствие в Go поддержки арифметики с указателями, благодаря чему ликвидируется целый класс потенциальных ошибок, часто проявляющихся при программировании на языках C и C++. Кроме того, в программах на языке Go не требуется вызывать функцию `free()` или использовать инструкцию `delete` благодаря наличию механизма *сборки мусора* и поддержки автоматического управления памятью¹. Значения ссылочных типов в языке Go создаются простым и уникальным способом, а после создания используются подобно ссылкам на объекты в языках Java и Python. Значения в языке Go действуют подобно значениям в других распространенных языках программирования.

Другие разделы этой главы посвящены встроенным типам коллекций. Здесь охватываются все встроенные типы коллекций: массивы, срезы и отображения. Эти типы являются настолько гибкими и эффективными, что способны удовлетворить практически любые потребности. В стандартной библиотеке имеются дополнительные, более специализированные типы коллекций: `container/heap`, `container/list` и `container/ring`, — обеспечивающие более высокую эффективность в определенных ситуациях. В последней главе представлена пара коротких примеров, демонстрирующих применение значений типов `heap` и `list` (§9.4.3). А в главе 6 приводится пример, демонстрирующий создание сбалансированного двоичного дерева (§6.5.3).

¹ Функция `delete()` в языке Go используется для удаления ключей из отображений, как будет показано ниже в этой главе.

4.1. Значения, указатели и ссылочные типы

В этом разделе будет обсуждаться то, что может храниться в переменных (значения, указатели и ссылки, включая ссылки на массивы, срезы и отображения), а фактическое использование массивов, срезов и отображений будет рассматриваться в следующих разделах.

Вообще говоря, переменные хранят *значения*. То есть переменную можно представлять себе как значение, хранящееся в ней. Исключение составляют переменные, ссылающиеся на каналы, функции, методы, отображения и срезы (они хранят *ссылки*), и переменные, хранящие указатели.

При передаче значений функциям или методам создаются их копии. Это совсем не дорого, когда речь идет о передаче логических или числовых значений, потому что каждое из них занимает от одного до восьми байт. Передача строк по значению – тоже достаточно недорогая операция, потому что компиляторы Go способны оптимизировать передачу так, что фактически будет передаваться очень небольшой объем данных, независимо от длины строки. Это обусловлено неизменяемостью строк в языке Go. (При передаче каждой строки фактически передаются всего 16 байт на 64-битных архитектурах и 8 байт на 32-битных архитектурах¹.) Конечно, если переданная строка изменяется внутри функции или метода (например, с помощью оператора `+=`), компилятор Go вынужден будет предусмотреть операцию *копирования при записи*, потенциально дорогостоящую для больших строк, но эту цену пришлось бы заплатить в любом случае, независимо от используемого языка.

В отличие от массивов в C или C++, *массивы* в языке Go передаются по значению, поэтому передача больших массивов является довольно дорогостоящей операцией. К счастью, массивы редко требуются в программах на языке Go, поскольку вместо них обычно используются *срезы*, как будет показано в следующем разделе. Цена передачи среза практически такая же, как цена передачи строки (то есть 16 байт на 64-битных архитектурах и 12 байт – на 32-битных), и

¹ Таковыми были объемы передаваемых данных на момент написания этих строк. Эти объемы зависят от внутренних особенностей реализации, но они никогда не будут слишком большими.

Инструкция	Переменная	Значение	Тип	Адрес в памяти
<code>y := 1.5</code>	y	1.5	float64	0xf8400000f8
<code>y++</code>	y	2.5	float64	0xf8400000f8
<code>z := math.Ceil(y)</code>	y	2.5	float64	0xf8400000f8
	x	2.5	float64	Изменяемая копия переменной y в функции Ceil()
	z	3.0	float64	0xf8400000c0

Рис. 4.1. Простые значения в памяти

не зависит от длины среза или его емкости¹. Однако при изменении среза внутри функции или метода никаких операций копирования при записи не выполняется, потому что, в отличие от строк, срезы являются изменяемыми значениями (то есть при изменении среза эти изменения будут отражаться на всех переменных, ссылающихся на него).

Рисунок 4.1 иллюстрирует взаимосвязь между переменными и памятью, которую они занимают. Серым цветом показаны адреса в памяти, потому что они могут изменяться от случая к случаю, а жирный шрифт используется для выделения изменений.

Концептуально *переменная* – это имя, присвоенное области памяти, где хранится значение определенного типа. То есть, если в программе имеется инструкция сокращенного объявления переменной `y := 1.5`, компилятор выделит область памяти, достаточную для хранения значения типа `float64` (то есть 8 байт), и сохранит в ней значение 1.5. С этого момента, пока поток выполнения не покинет область видимости переменной `y`, это имя будет интерпретироваться компилятором как синоним адреса в памяти, где хранится значение типа `float64`, связанное с переменной `y`. Если за объявлением следует инструкция `y++`, компилятор Go увеличит значение, связанное с переменной `y`. Однако если передать переменную `y` функции или методу, компилятор передаст копию `y`, другими словами, компилятор создаст новую переменную, связанную с соответствующим параметром вызываемой функции или метода, и скопирует значение переменной `y` в память, выделенную для новой переменной.

Иногда бывает необходимо, чтобы функция имела возможность изменить значение переданной ей переменной. Реализовать это можно с помощью ссылки, как будет показано ниже, но при передаче

¹ Таковыми были объемы передаваемых данных на момент написания этих строк. Эти объемы зависят от внутренних особенностей реализации, но они никогда не будут слишком большими.

параметра по значению функция получит копию, и все изменения будут применяться к копии, а оригинальное значение останется неизменным. Кроме того, передача копированием некоторых значений (таких как массивы или структуры с большим количеством полей), занимающих большие объемы памяти, является дорогостоящей операцией. Помимо этого, локальные переменные утилизируются сборщиком мусора, как только они становятся ненужными (например, когда не остается ни одной ссылки на них и поток выполнения выходит из области их видимости), однако во многих ситуациях бывает желательно создавать переменные, срок жизни которых определяется программистом, а не их областью видимости.

Избежать дорогостоящего копирования параметров при передаче, обеспечить возможность изменения значений параметров в функциях и самостоятельно определять сроки жизни переменных независимо от области их видимости можно за счет использования указателей. *Указатель* – это переменная, хранящая адрес значения другой переменной. При создании указателя запоминается тип переменной, на которую он указывает, – это позволяет компилятору Go определять объем памяти (то есть количество байтов), занимаемый значением, на которое указывает указатель. Значение переменной, на которую указывает указатель, может изменяться с помощью этого указателя, как будет показано чуть ниже. Стоимость передачи указателя невелика (8 байт на 64-битных архитектурах и 4 байта на 32-битных) и не зависит от размера значения, на которое он указывает. Кроме того, переменная, на которую указывает указатель, может сохраняться до тех пор, пока существует хотя бы один указатель, указывающий на нее, что обеспечивает возможность продления срока жизни переменной независимо от области видимости, где она была создана¹.

Оператор `&` в языке Go перегружен. Когда он используется как двухместный оператор, выполняется операция «поразрядное И». Когда он используется как унарный оператор, возвращается адрес операнда в памяти – именно этот адрес можно использовать для сохранения в указателе. В третьей инструкции на рис. 4.2 выполняется присваивание адреса переменной `x` типа `int`, переменной `pi`, имеющей тип `*int` (указатель на значение типа `int`). Унарный

¹ Специально для программистов на C и C++: несмотря на то что компиляторы языка Go способны различать стек и обычную память, программистам на Go нет необходимости волноваться об этом, так как все управление памятью берет на себя Go.

Инструкция	Переменная	Значение	Тип	Адрес в памяти
x := 3	x	3	int	0xf840000148
y := 22	y	22	int	0xf840000150
x == 3 && y == 22				
	x	3	int	0xf840000148
pi := &x	pi	0xf840000148	*int	0xf840000158
*pi == 3 && x == 3 && y == 22				
	x	4	int	0xf840000148
x++	pi	0xf840000148	*int	0xf840000158
*pi == 4 && x == 4 && y == 22				
	x	5	int	0xf840000148
*pi++	pi	0xf840000148	*int	0xf840000158
*pi == 5 && x == 5 && y == 22				
	y	y	int	0xf840000150
pi := &y	pi	0xf840000150	*int	0xf840000158
*pi == 22 && x == 5 && y == 22				
	y	23	int	0xf840000150
*pi++	pi	0xf840000150	*int	0xf840000158
*pi == 23 && x == 5 && y == 23				

Рис. 4.2. Указатели и значения

оператор & иногда называют оператором *взятия адреса*. Под термином *указатель* подразумевается, что переменная, хранящая адрес другой переменной, «указывает на» другую переменную, как показано стрелками на рис. 4.2.

Оператор * также перегружен. Когда он используется как двухместный оператор, выполняется операция умножения. А когда он используется как унарный оператор, обеспечивается доступ к значению, на которое указывает переменная-операнд. То есть, как показано на рис. 4.2, после выполнения инструкции `pi := &x` (но до того, как переменной `pi` будет присвоен адрес другой переменной) `*pi` и `x` могут использоваться взаимозаменяемо. А поскольку обе они связаны с одним и тем же целочисленным значением в памяти, любые изменения, произведенные с использованием одной переменной, будут оказывать влияние на другую переменную. Унарный оператор * иногда называют оператором *доступа к содержимому, косвенной адресации* или *разыменования указателя*.

На рис. 4.2 также видно, что при изменении переменной, на которую ссылается указатель (например, в результате выполнения инструкции `x++`), ее значение изменится, как и следовало бы ожидать,

и операция разыменования указателя на нее (`*pi`) вернет новое значение. Точно так же можно изменить значение с помощью указателя. Например, инструкция `*pi++` увеличит значение переменной, на которую ссылается указатель. Конечно, подобная инструкция может быть скомпилирована, только если тип значения поддерживает оператор `++`, как в случае с числами.

Указатель не обязательно должен все время ссылаться на одно и то же значение. Например, на рис. 4.2, в нижней его части, демонстрируется возможность присваивания указателю адреса другого значения (`pi := &y`) и последующего его изменения с помощью указателя. Это изменение легко можно было бы выполнить непосредственно через переменную `y` (например, с помощью инструкции `y++`) и получить новое значение переменной `y` с помощью выражения `*pi`.

Кроме того, существует возможность создавать указатели на указатели (а также указатели на указатели на указатели и т. д.). Использование указателя для ссылки на значение называется *косвенной адресацией*. А использование указателя на указатель называется *многоуровневой косвенной адресацией*. Такая форма адресации часто используется в языках C и C++, но в Go потребность в ней возникает достаточно редко, благодаря поддержке ссылочных типов. Например:

```
z := 37    // z имеет тип int
pi := &z   // pi имеет тип *int (указатель на значение типа int)
ppi := &pi // ppi имеет тип **int (указатель на указатель на значение типа int)
fmt.Println(z, *pi, **ppi)
**ppi++    // Эту операцию можно также представить как: (*(*ppi))++ или *(*ppi)++
fmt.Println(z, *pi, **ppi)
37 37 37
38 38 38
```

В данном фрагменте `pi` является указателем типа `*int` (указатель на значение типа `int`), то есть указателем на переменную `z` типа `int`, а `ppi` — указателем типа `**int` (указатель на указатель на значение типа `int`), то есть указателем на переменную `pi`. При разыменовании используется один оператор `*` для каждого уровня косвенности, то есть операция `*ppi` разыменует указатель `ppi` и вернет указатель типа `*int` — адрес в памяти, а применение второго оператора `*` (`**ppi`) даст в результате само значение типа `int`.

Помимо умножения и разыменования, оператор `*` может также выступать в роли модификатора типа. Когда оператор `*` помещается

слева от имени типа, он изменяет значение имени, в результате чего создается не значение указанного типа, а указатель на значение этого типа. Эта особенность демонстрируется в колонке «Тип» на рис. 4.2.

Ниже приводится короткий пример, демонстрирующий все, о чем говорилось выше.

```
i := 9
j := 5
product := 0
swapAndProduct1(&i, &j, &product)
fmt.Println(i, j, product)
5 9 45
```

Здесь создаются три переменные типа `int`, и им присваиваются начальные значения. Затем вызывается пользовательская функция `swapAndProduct1()`, принимающая три указателя на значения типа `int`, упорядочивающая первые два значения в порядке возрастания и возвращающая в третьем произведение первых двух. Так как функция принимает указатели, ей следует передавать адреса на значения типа `int`, а не сами эти значения. Всякий раз, увидев оператор взятия адреса `&` в вызове функции, можно смело предполагать, что значение соответствующей переменной может измениться внутри функции. Ниже приводится сама функция `swapAndProduct1()`.

```
func swapAndProduct1(x, y, product *int) {
    if *x > *y {
        *x, *y = *y, *x
    }
    *product = *x * *y // Эту же инструкцию можно записать как: *product=*x*y
}
```

Параметры функции объявлены как значения типа `*int`, где оператор `*` играет роль модификатора типа, определяя, что все параметры являются указателями на целые числа. Это также означает, что функции могут передаваться только адреса целочисленных переменных (с помощью оператора взятия адреса `&`), но не сами эти переменные и не целочисленные литералы.

Внутри функции требуется выполнить операции с самими значениями, на которые ссылаются указатели, поэтому здесь повсюду используется оператор разыменования `*`. В последней строке

выполняется умножение двух значений, на которые ссылаются указатели, и результат присваивается по другому указателю. Компилятор Go способен отличить последовательность операторов умножения и разыменования от двух операторов разыменования, исходя из контекста. Внутри функции указатели имеют имена `x`, `y` и `product`, но за пределами функции целочисленные значения, на которые они указывают, имеют имена `i`, `j` и `product`.

Подобный стиль реализации функций часто можно встретить в программном коде на языке C и в старом программном коде на C++, но он редко используется в языке Go. Когда требуется передать одно или несколько значений, в языке Go принято просто передавать эти значения. А когда требуется передать множество значений, они обычно передаются в виде среза или отображения (передача которых стоит недорого и без использования указателей, как будет показано чуть ниже), или в виде указателя на структуру, если значения имеют разные типы. Ниже приводится альтернативная реализация без использования указателей:

```
i := 9
j := 5
i, j, product := swapAndProduct2(i, j)
fmt.Println(i, j, product)
5 9 45
```

И соответствующая функция `swapAndProduct2()`.

```
func swapAndProduct2(x, y int) (int, int, int) {
    if x > y {
        x, y = y, x
    }
    return x, y, x * y
}
```

Эта версия функции выглядит проще, но без указателей она не может выполнить перестановку переменных на месте.

В программах на C и C++ часто можно встретить функции, принимающие указатель на логическое значение, которое служит индикатором успеха или неудачи. То же самое легко можно реализовать на языке Go, добавив в сигнатуру функции параметр типа `*bool`, но гораздо удобнее просто возвращать логический флаг успеха (или, еще лучше, значение ошибки) в качестве последнего (или единственного) возвращаемого значения, что является стандартной практикой при программировании на языке Go.

Во фрагментах программного кода, представленных выше, для получения адресов параметров функций или локальных переменных использовался оператор взятия адреса `&`. Благодаря механизму автоматического управления памятью в языке Go использование этих адресов не таит в себе неожиданностей, потому что пока на переменную ссылается хотя бы один указатель, она будет продолжать храниться в памяти. Именно поэтому в языке Go можно спокойно возвращать указатели на локальные переменные, созданные внутри функции (что может привести к фатальным последствиям в языках C и C++ в случае нестатических переменных).

В ситуациях, когда необходимо передавать для изменения значения не ссылочных типов или значения, занимающие большой объем, для повышения эффективности может потребоваться использовать указатели. В языке Go имеется возможность одновременно создавать переменные и получать указатели на них. Первая из них заключается в использовании встроенной функции `new()`, а вторая – в использовании оператора взятия адреса. Рассмотрим оба способа на примере создания значений простого составного типа.

```
type composer struct {  
    name string  
    birthYear int  
}
```

Имея такое определение структуры, можно создавать значения типа `composer` или указатели на значения типа `composer`, то есть переменные типа `*composer`. И в обоих случаях использовать поддержку инициализации структур с применением фигурных скобок.

```
antônio := composer{"Antônio Teixeira", 1707} // значение типа composer  
agnes := new(composer) // указатель на значение типа composer  
agnes.name, agnes.birthYear = "Agnes Zimmermann", 1845  
julia := &composer{} // указатель на значение типа composer  
julia.name, julia.birthYear = "Julia Ward Howe", 1819  
augusta := &composer{"Augusta Holmès", 1847} // указатель на значение типа composer  
fmt.Println(antônio)  
fmt.Println(agnes, augusta, julia)  
{Antônio Teixeira 1707}  
&{Agnes Zimmermann 1845} &{Augusta Holmès 1847} &{Julia Ward Howe 1819}
```

При выводе указателей на структуры компилятор Go размынчивает их, но добавляет в начало оператор взятия адреса &, чтобы показать, что это указатель. В части фрагмента, где создаются указатели `agnes` и `julia`, демонстрируется справедливость следующего тождества, когда значение типа может инициализироваться с помощью фигурных скобок:

```
new(Тип) == &Тип{}
```

В обоих случаях создается новое пустое значение указанного Типа и возвращается указатель на это значение. Если Тип не является типом, который можно инициализировать с помощью фигурных скобок, в этом случае допускается использовать только прием на основе встроенной функции `new()`. И конечно же нет никакой необходимости задумываться об управлении жизненным циклом значения или удалять его явно, потому что обо всем позаботится система автоматического управления памятью языка Go.

Одно из преимуществ использования синтаксической конструкции `&Тип{}` для создания значений структур состоит в том, что она позволяет указать начальные значения полей, как это показано в инструкции создания указателя `augusta`. (Эта конструкция дает возможность даже указать начальные значения лишь для отдельных полей, как будет показано ниже, в §6.4.)

Вдобавок к значениям и указателям в языке Go имеются *ссылочные типы*. (В Go также имеются интерфейсы, но с практической точки зрения интерфейсы можно считать разновидностью ссылочных типов; интерфейсы будут рассматриваться далее, в §6.3.) Переменная ссылочного типа ссылается на скрытое значение в памяти, где хранятся фактические данные. Переменные ссылочных типов недороги при передаче (например, 16 байт для срезов и 8 байт для отображений на 64-битных архитектурах) и используются идентично обычным значениям (то есть для доступа к фактическому значению не требуется получать адрес значения ссылочного типа или размынчивать его).

Если из функции или метода потребуется возвращать более четырех-пяти значений, для этой цели лучше всего использовать срез, если все значения имеют один и тот же тип, или указатель на структуру, если значения имеют разные типы. Передача среза или указателя на структуру стоит недорого и позволяет изменять данные на месте. Рассмотрим пару небольших примеров, иллюстрирующих эти возможности.

```
grades := []int{87, 55, 43, 71, 60, 43, 32, 19, 63}
inflate(grades, 3)
fmt.Println(grades)
[261 165 129 213 180 129 96 57 189]
```

Здесь выполняется операция над всеми числами в срезе. Отображения и срезы относятся к ссылочным типам, и любые изменения в элементах отображения или среза, выполненные непосредственно или внутри функции, которой они передаются, будут отражаться на всех переменных, ссылающихся на них.

```
func inflate(numbers []int, factor int) {
    for i := range numbers {
        numbers[i] *= factor
    }
}
```

Срез `grades` передается в параметре с именем `numbers`, но, в отличие от обычных значений, любые изменения, выполненные в срезе под именем `numbers`, будут отражаться на срезе под именем `grades`, поскольку оба имени ссылаются на один и тот же срез в памяти.

Здесь требуется изменить значения элементов среза на месте, поэтому доступ к ним осуществляется с помощью счетчика цикла. В данном случае нельзя использовать цикл `for index, item ...range`, потому что в теле цикла будут доступны лишь копии элементов среза и умножаться на значение `factor` будут копии, а оригинальный срез останется неизменным. Можно было бы использовать цикл `for` в виде, в каком он используется в других языках (например, `for i := 0; i < len(numbers); i++`), но здесь было отдано предпочтение более удобной его разновидности `for index := range`. (Все имеющиеся разновидности цикла `for` рассматриваются в следующей главе, в §5.3.)

Теперь представьте, что имеется некоторый тип `rectangle`, значения которого хранят координаты верхнего левого и правого нижнего углов прямоугольника, а также цвет заливки. Данные о прямоугольнике можно было бы представить в виде структуры:

```
type rectangle struct {
    x0, y0, x1, y1 int
    fill           color.RGBA
}
```

Далее необходимо создать значение типа `rectangle`, вывести его, изменить размеры прямоугольника и снова вывести.

```
rect := rectangle(4, 8, 20, 10, color.RGBA{0xFF, 0, 0, 0xFF})
fmt.Println(rect)
resizeRect(&rect, 5, 5)
fmt.Println(rect)
{4 8 20 10 {255 0 0 255}}
{4 8 25 15 {255 0 0 255}}
```

Как отмечалось в предыдущей главе, даже при том, что тип `rectangle` является пользовательским типом, значения этого типа все равно могут быть выведены с помощью стандартных функций вывода. Вывод, показанный под фрагментом программного кода, наглядно демонстрирует, что пользовательская функция `resizeRect()` с успехом справилась со своим заданием. И вместо передачи всего значения типа `rectangle` (16 байт – для одних только полей типа `int`) оказалось достаточным передать лишь его адрес (8 байт на 64-битных архитектурах, независимо от размера самой структуры).

```
func resizeRect(rect *rectangle, Δwidth, Δheight int) {
    (*rect).x1 += Δwidth // Неуклюжий способ явного разыменования
    rect.y1 += Δheight   // Точка (.) автоматически разыменовывает указатели
}                        // на структуры
```

В первой инструкции здесь используется операция явного разыменования, но только чтобы показать, что в действительности происходит за кулисами. Фрагмент `(*rect)` соответствует фактическому значению типа `rectangle`, на которое ссылается указатель `rect`, а фрагмент `.x1` соответствует полю `x1`. Вторая инструкция демонстрирует более типичный способ работы со структурами (или с указателями на структуры). В последнем случае разыменовывание указателя в языке Go выполняется автоматически. Это обусловлено тем, что оператор выбора `.` (точка) автоматически разыменовывает указатели на структуры¹.

¹ В языке Go отсутствует и не нужен оператор разыменования `->`, имеющийся в языках C и C++. Для большинства ситуаций достаточно оператора точки `.` (например, для доступа к полям структуры или указателя на структуру), но там, где это невозможно, можно использовать операцию явного разыменования, употребляя столько операторов `*`, сколько имеет-ся уровней косвенности.

Некоторые типы в языке, такие как отображения, срезы, каналы, функции и методы, являются ссылочными. В отличие от указателей, для работы со значениями ссылочных типов не требуется использовать специальный синтаксис – они действуют подобно обычным значениям. В программах допускается использовать указатели на значения ссылочных типов, хотя в действительности такая возможность может пригодиться (а иногда просто необходима) только при работе со срезами. (Пример использования указателя на срез будет представлен в следующей главе, в §5.7.)

При объявлении переменной для хранения функции этой переменной в действительности присваивается ссылка на функцию. Для *ссылок на функции* известны сигнатуры функций, на которые они ссылаются, поэтому невозможно передать ссылку на функцию с другой сигнатурой. Благодаря этому устраняются по-настоящему досадные ошибки, которые могут возникать в языках, допускающих передачу функций по указателям, но не гарантирующих, что такие функции будут иметь требуемую сигнатуру. Выше уже встречалось несколько примеров передачи ссылок на функции, например когда функции `strings.Map()` передавалась функция отображения (§3.6 и §3.7). Далее в книге будет представлено еще множество примеров использования указателей и ссылочных типов.

4.2. Массивы и срезы

Массивы в языке Go – это последовательности элементов одного типа фиксированной длины. *Многомерные массивы* могут создаваться за счет использования элементов, которые сами являются массивами.

Доступ к элементам массива осуществляется с помощью оператора индексирования `[]`, где отсчет индексов начинается с 0, то есть первый элемент массива имеет индекс `array[0]`, а последний – `array[len(array) - 1]`. Массивы являются *изменяемыми* значениями, то есть для изменения значения элемента массива `array` в позиции `index` слева от оператора присваивания можно указать выражение `array[index]`. То же самое выражение можно указать справа от оператора присваивания или в вызове функции, чтобы обеспечить доступ к элементу.

Массивы создаются следующим образом:

`[длина]Тип`

`[N]Тип{значение1, значение2, ..., значениеN}`

`[...]Тип { значение1, значение2, ..., значениеN }`

Если здесь задействовать оператор `...` (многоточие), компилятор Go вычислит размер массива автоматически. (Оператор многоточия перегружен и может выполнять несколько операций, как будет показано далее в этой главе и в главе 5.) В любом случае массив имеет фиксированную длину, которую невозможно изменить.

Ниже приводятся несколько примеров, демонстрирующих способы создания массивов и доступа к их элементам.

```
var buffer [20]byte
var grid1 [3][3]int
grid1[1][0], grid1[1][1], grid1[1][2] = 8, 6, 2
grid2 := [3][3]int{{4, 3}, {8, 6, 2}}
cities := [...]string{"Shanghai", "Mumbai", "Istanbul", "Beijing"}
cities[len(cities)-1] = "Karachi"
fmt.Println("Type      Len Contents")
fmt.Printf("%-8T %2d %v\n", buffer, len(buffer), buffer)
fmt.Printf("%-8T %2d %q\n", cities, len(cities), cities)
fmt.Printf("%-8T %2d %v\n", grid1, len(grid1), grid1)
fmt.Printf("%-8T %2d %v\n", grid2, len(grid2), grid2)
Type      Len Contents
[20]uint8 20 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[4]string  4 ["Shanghai" "Mumbai" "Istanbul" "Karachi"]
[3][3]int  3 [[0 0 0] [8 6 2] [0 0 0]]
[3][3]int  3 [[4 3 0] [8 6 2] [0 0 0]]
```

В языке Go гарантируется, что все элементы массива будут инициализированы соответствующими нулевыми значениями, если они не будут инициализированы явно при создании, как в случае с переменными `buffer`, `grid1` и `grid2` в примере выше.

Длину массива можно получить с помощью функции `len()`. Поскольку массивы имеют фиксированную длину, их емкость всегда равна их длине, поэтому для массивов функция `cap()` всегда возвращает то же значение, что и функция `len()`. Из массива можно извлечь срез, используя тот же синтаксис извлечения срезов, как и в случае со строками или срезами, только результатом такой операции будет срез, а не массив. И подобно строкам и срезам, итерации по массивам можно выполнять с помощью цикла `for ...range` (§5.3).

В целом *срезы* в языке Go более гибкие, более мощные и более удобные, чем массивы. Массивы передаются по значению (то есть копируются), хотя этого можно избежать с помощью указателей, тогда как срезы всегда передаются по ссылке, независимо от их длины

и емкости. (Срез передается как 16-байтовое значение на 64-битных архитектурах и 12-байтовое – на 32-битных архитектурах, независимо от количества элементов, содержащихся в них.) Массивы имеют фиксированный размер, тогда как размеры срезов могут изменяться. Все функции в стандартной библиотеке Go, образующие общедоступный API, используют срезы, а не массивы¹. Если нет каких-то определенных требований, всегда рекомендуется использовать срезы. И массивы, и срезы поддерживают операцию извлечения среза с использованием синтаксиса, представленного в табл. 4.1 (ниже).

Срезы – это последовательности элементов одного типа фиксированной емкости и переменной длины. Несмотря на фиксированную емкость, срезы могут укорачиваться путем их усечения и удлиняться с помощью эффективной встроенной функции `append()`, как будет показано далее в этом разделе. *Многомерные срезы* можно создавать, используя элементы, которые сами являются срезами, при этом длины внутренних срезов в многомерном срезе могут быть разными.

Несмотря на требование принадлежности элементов массивов и срезов к одному типу, на практике никаких ограничений не существует. Это обусловлено тем, что роль типа здесь может играть интерфейс. Поэтому элементы могут быть любого типа, при условии что все они поддерживают указанный интерфейс (то есть имеют метод или методы, определяемые интерфейсом). Можно даже создать массив или срез с типом пустого интерфейса, `interface{}`, и хранить в нем элементы любых типов, однако при обращении к ним придется использовать инструкции приведения типа и выбора по типу или механизм интроспекции, чтобы обеспечить корректное их использование. (Интерфейсы подробно рассматриваются в главе 6; механизм рефлексии – в §9.4.9.)

Для создания срезов используется следующий синтаксис:

```
make([]Тип, длина, емкость)
make([]Тип, длина)
[]Тип {}
[]Тип {значение1, значение2, ..., значениеN}
```

Встроенная функция `make()` используется для создания срезов, отображений и каналов. Когда она применяется для создания среза,

¹ На момент написания этих строк в официальной документации к языку Go, в описаниях параметров функций, которые в действительности являются срезами, часто используется термин *массив* (*array*).

то создает *скрытый массив* с элементами, инициализированными нулевыми значениями, и возвращает ссылку на срез, ссылающуюся на скрытый массив. Скрытый массив, как и все массивы в языке Go, имеет фиксированную длину, равную емкости среза, если для его создания использован первый способ, или длине, если использован второй способ, или числу элементов в фигурных скобках, если использован составной литерал (третий и четвертый способы).

Емкость среза – это длина скрытого массива, а длина среза – любое количество элементов от нуля до значения емкости. При создании среза первым способом указанная длина должна быть меньше или равна емкости, хотя обычно этот способ используется, когда необходимо, чтобы первоначальная длина среза была меньше его емкости. Второй, третий и четвертый способы используются, когда желательно, чтобы длина совпадала с емкостью. Способ на основе составного литерала (четвертый) удобно использовать для создания среза с некоторыми начальными значениями.

Таблица 4.1. Операции, поддерживаемые срезами

Операция	Описание/результат
<code>s[n]</code>	Элемент в позиции <code>n</code> в срезе <code>s</code>
<code>s[n:m]</code>	Срез из среза <code>s</code> , начиная с элемента в позиции <code>n</code> и до элемента в позиции <code>m-1</code> включительно
<code>s[n:]</code>	Срез из среза <code>s</code> , начиная с элемента в позиции <code>n</code> и до последнего элемента в позиции <code>len(s) - 1</code> включительно
<code>s[:m]</code>	Срез из среза <code>s</code> , начиная с элемента в позиции <code>0</code> и до элемента в позиции <code>m-1</code> включительно
<code>s[:]</code>	Срез из среза <code>s</code> , начиная с элемента в позиции <code>0</code> и до последнего элемента в позиции <code>len(s) - 1</code> включительно
<code>cap(s)</code>	Емкость среза <code>s</code> ; всегда больше или равна <code>len(s)</code>
<code>len(s)</code>	Количество элементов в срезе <code>s</code> ; всегда меньше или равно <code>cap(s)</code>
<code>s = s[:cap(s)]</code>	Увеличит длину среза <code>s</code> до его емкости, если они не равны

Синтаксическая конструкция `[]Тип{}` эквивалентна вызову функции `make([]Тип, 0)` – обе создают пустой срез. Этот способ создания срезов не является бесполезным, как могло бы показаться, благодаря возможности использовать встроенную функцию `append()` для увеличения емкости среза. Однако на практике, когда требуется пустой срез, почти всегда лучше создавать его вызовом функции `make()`, указывая нулевую длину и ненулевую емкость, примерно равную ожидаемому числу элементов.

Допустимыми значениями индексов срезов являются целые числа в диапазоне от 0 до $\text{len}(\text{срез}) - 1$. Длину среза можно уменьшить операцией усечения, а если емкость среза больше его длины, длину можно увеличить до значения емкости. Имеется также возможность увеличивать емкость срезов с помощью встроенной функции `append()` – примеры ее использования будут представлены ниже в этом разделе.

На рис. 4.3 приводится концептуальное представление взаимосвязи между срезами и их скрытыми массивами. На рисунке изображен следующий срез:

```
s := []string{"A", "B", "C", "D", "E", "F", "G"}
t := s[:5]           // [A B C D E]
u := s[3 : len(s)-1] // [D E F]
fmt.Println(s, t, u)
u[1] = "x"
fmt.Println(s, t, u)
[A B C D E F G] [A B C D E] [D E F]
[A B C D x F G] [A B C D x] [D x F]
```

Поскольку все переменные `s`, `t` и `u` ссылаются на одни и те же данные в памяти, изменение одной влечет изменение всех остальных, ссылающихся на те же данные.

```
s := new([7]string)[: ]
s[0], s[1], s[2], s[3], s[4], s[5], s[6] = "A", "B", "C", "D", "E", "F", "G"
```

Лучшими способами создания срезов являются встроенная функция `make()` и синтаксис составных литералов, а здесь показан способ, который не используется на практике, но проясняет связь между массивами и срезами. Первая инструкция создает указатель на массив вызовом встроенной функции `new()` и тут же извлекает срез, включающий весь массив. В результате создается срез, длина

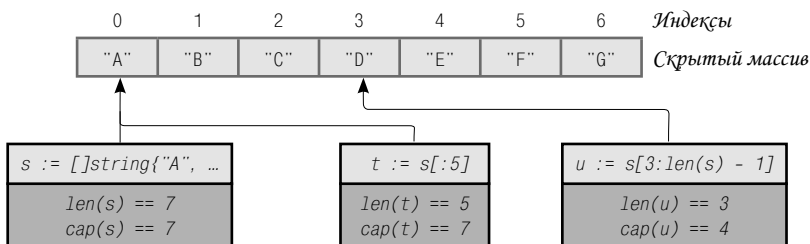


Рис. 4.3. Взаимосвязи между срезами и их скрытыми массивами

и емкость которого равны длине массива и каждому элементу присвоено нулевое значение – в данном случае пустая строка. Вторая инструкция завершает создание среза, заполняя элементы желаемыми начальными значениями, после чего данный срез `s` становится точно таким же, как созданный с помощью синтаксиса составных литералов в предыдущем фрагменте.

Ниже приводится пример на основе срезов, эквивалентный примеру выше, демонстрирующему массивы. Они отличаются лишь тем, что здесь емкость среза `buffer` выбрана больше его длины, чтобы показать, как это делается.

```
buffer := make([]byte, 20, 60)
grid1 := make([]int, 3)
for i := range grid1 {
    grid1[i] = make(int, 3)
}
grid1[1][0], grid1[1][1], grid1[1][2] = 8, 6, 2
grid2 := [][]int{{4, 3, 0}, {8, 6, 2}, {0, 0, 0}}
cities := []string{"Shanghai", "Mumbai", "Istanbul", "Beijing"}
cities[len(cities)-1] = "Karachi"
fmt.Println("Type   Len Cap Contents")
fmt.Printf("%-8T %2d %3d %v\n", buffer, len(buffer), cap(buffer), buffer)
fmt.Printf("%-8T %2d %3d %q\n", cities, len(cities), cap(cities), cities)
fmt.Printf("%-8T %2d %3d %v\n", grid1, len(grid1), cap(grid1), grid1)
fmt.Printf("%-8T %2d %3d %v\n", grid2, len(grid2), cap(grid2), grid2)
Type   Len Cap Contents
[]uint8 20 60 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[]string 4   4 ["Shanghai" "Mumbai" "Istanbul" "Karachi"]
[][]int  3   3 [[0 0 0] [8 6 2] [0 0 0]]
[][]int  3   3 [[4 3 0] [8 6 2] [0 0 0]]
```

Содержимое среза `buffer` – это лишь первые `len(buffer)` элементов. Остальные элементы недоступны, пока длина среза не будет увеличена, как будет показано ниже в этом разделе.

Срез `grid1` создан как срез срезов с начальной длиной 3 (то есть он может содержать три среза) и емкостью 3 (так как по умолчанию емкость устанавливается равной длине, если явно не указано другое значение). Затем каждому элементу среза `grid` присваивается свой срез из трех элементов. Естественно, при необходимости внутренние срезы могли бы иметь разные длины.

Срез `grid2` создается с использованием синтаксиса составных литералов, и для него указываются значения всех его элементов,

так как в этом случае компилятор Go не имеет другой возможности определить желаемое количество элементов. Здесь также внутренние срезы можно было бы создать с разными длинами, например `grid2 := [][]int{{9, 7}, {8}, {4, 2, 6}}`, и в результате получить срез `grid2` с длиной 3, содержащий срезы с длинами 2, 1 и 3.

4.2.1. Индексирование срезов и извлечение срезов из срезов

Срез – это ссылка на скрытый массив. И срез, извлеченный из среза, ссылается на тот же самый скрытый массив. Ниже приводится пример, поясняющий эти слова.

```
s := []string{"A", "B", "C", "D", "E", "F", "G"}
t := s[2:6]
fmt.Println(t, s, "=", s[:4], "+", s[4:])
s[3] = "x"
t[len(t)-1] = "y"
fmt.Println(t, s, "=", s[:4], "+", s[4:])
[C D E F] [A B C D E F G] = [A B C D] + [E F G]
[C x E y] [A B C x E y G] = [A B C x] + [E y G]
```

При изменении данных посредством оригинальной переменной `s` или переменной `t`, полученной в результате извлечения среза из среза, изменяются одни и те же данные в памяти, то есть изменения затрагивают оба среза. Данный фрагмент также демонстрирует, что для данного среза `s` и индекса `i` ($0 \leq i \leq \text{len}(s)$) срез `s` равен конкатенации срезов `s[:i]` и `s[i:]`. Подобное тождество встречалось нам в предыдущей главе применительно к строкам:

```
s == s[:i] + s[i:] // s - это строка; ; i - значение типа int; 0 <= i <= len(s)
```

На рис. 4.4 показан срез `s`, включая все допустимые позиции, и срезы, извлекаемые в фрагменте программного кода. Нумерация позиций в любом срезе начинается с 0, а последний элемент среза всегда имеет индекс `len(s) - 1`.

В отличие от строк, срезов не поддерживают операторы `+` и `+=`. Тем не менее добавление в конец, а также вставка и удаление элементов выполняются довольно просто, как будет показано чуть ниже (§4.2.3).

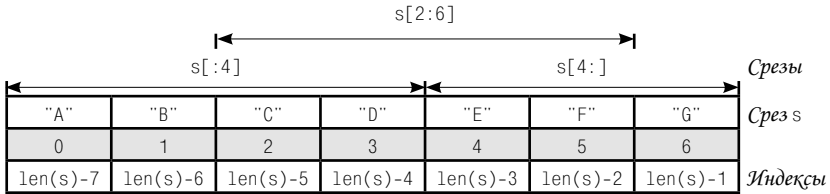


Рис. 4.4. Устройство среза

4.2.2. Итерации по срезам

На практике часто возникает необходимость выполнить *итерации* по всем элементам среза. Если в ходе итераций не требуется изменять значения элементов, можно воспользоваться циклом `for ...range`, в противном случае следует использовать цикл `for` со счетчиком. Ниже приводится пример реализации первого случая.

```
amounts := []float64{237.81, 261.87, 273.93, 279.99, 281.07, 303.17,
    231.47, 227.33, 209.23, 197.09}
sum := 0.0
for _, amount := range amounts {
    sum += amount
}
fmt.Printf("Σ %.1f → %.1f\n", amounts, sum)
Σ [237.8 261.9 273.9 280.0 281.1 303.2 231.5 227.3 209.2 197.1] → 2503.0
```

Цикл `for ...range` имеет две переменные цикла: счетчик итераций, начинающий отсчет с 0, который в этом примере не используется и отбрасывается присваиванием *пустому идентификатору* (`_`), и *копия* соответствующего элемента среза. Копирование не влечет больших накладных расходов, даже для строк (так как они передаются по ссылке). То есть любые изменения в полученном элементе будут затрагивать только копию, а не сам элемент в срезе.

Естественно, для обхода части среза можно использовать операцию извлечения среза. Например, если потребуется выполнить итерации по первым пяти элементам, цикл можно записать так: `for _, amount := range amounts[:5]`.

Если в цикле необходимо иметь возможность изменять значения элементов, следует использовать цикл `for`, поставляющий только допустимые индексы, но не копии элементов в срезе.

```

for i := range amounts {
    amounts[i] *= 1.05
    sum += amounts[i]
}
fmt.Printf("Σ %.1f → %.1f\n", amounts, sum)
Σ [249.7 275.0 287.6 294.0 295.1 318.3 243.0 238.7 219.7 206.9] → 2628.1

```

Здесь каждый элемент среза увеличивается на 5%, и накапливается общая сумма новых значений.

Срезы могут также содержать элементы пользовательских типов. Ниже приводится определение пользовательского типа с единственным методом.

```

type Product struct {
    name string
    price float64
}
func (product Product) String() string {
    return fmt.Sprintf("%s (%.2f)", product.name, product.price)
}

```

Здесь определяется тип `Product` как структура с полями типа `string` и `float64`. Тут также определяется метод `String()`, управляющий выводом значений типа `Product` с использованием спецификатора `%v`. (Спецификаторы вывода обсуждаются выше, в §3.5. Краткое введение в пользовательские типы и методы приводится в §1.5 – подробнее об этом рассказывается в главе 6.)

```

products := []*Product{{"Spanner", 3.99}, {"Wrench", 2.49},
    {"Screwdriver", 1.99}}
fmt.Println(products)
for _, product := range products {
    product.price += 0.50
}
fmt.Println(products)
[Spanner (3.99) Wrench (2.49) Screwdriver (1.99)]
[Spanner (4.49) Wrench (2.99) Screwdriver (2.49)]

```

Здесь создается срез указателей на значения типа `Product` (`[]*Product`) и сразу же инициализируется тремя значениями `*Product`. Такое возможно благодаря тому, что компилятор Go способен понять, что элементами среза типа `[]*Product` являются указатели

на значения типа `Product`. Данная запись является сокращенной формой объявления: `products := []*Product{&Product{"Spanner", 3.99}, &Product{"Wrench", 2.49}, &Product{"Screwdriver", 1.99}}`. (В §4.1 уже говорилось, что синтаксическая конструкция `&Тип{}` создает новое значение указанного типа и возвращает указатель на него.)

Если бы в примере выше не был объявлен метод `Product.String()`, при выводе значений этого типа с применением спецификатора `%v` (который неявно используется функцией `fmt.Println()` и подобными ей) просто выводились бы адреса значений типа `Product`, а не сами значения. Отметьте также, что метод `Product.String()` принимает аргумент типа `Product`, а не `*Product`, однако в этом нет никакой проблемы, потому что компилятор Go автоматически будет разыменовывать значения `*Product` для передачи пользовательскому методу¹.

Выше отмечалось, что цикл `for ... range` не может использоваться в ситуациях, когда требуется иметь возможность изменять значения элементов среза во время итераций. Однако в этом примере для всех элементов среза удалось благополучно увеличить значения полей `price`. В каждой итерации переменной `product` присваивается копия значения типа `*Product` – то есть копия указателя из соответствующего элемента среза, ссылающегося на фактическое значение типа `Product` в памяти. Поэтому изменения применяются к значению `Product`, на которое указывает указатель, а не к копии указателя `*Product`.

4.2.3. Изменение срезов

Для добавления новых значений в конец среза можно использовать встроенную функцию `append()`. Эта функция принимает срез и одно или более значений добавляемых *отдельных* элементов. При необходимости добавить содержимое другого среза следует использовать оператор `...` (многоточие), чтобы сообщить компилятору Go, что содержимое указанного среза должно добавляться поэлементно. Добавляемые значения должны иметь тот же тип, что и сам срез. При работе со строками оператор многоточия можно использовать для добавления отдельных байтов из строки в срез с байтами.

¹ Компилятор делает это следующим образом: когда определяется метод, оперирующий значением, пусть это будет метод `Method()`, автоматически создается метод-обертка с тем же именем и сигнатурой, приемником в котором является указатель `func (value *Type) Method() { return (*value).Method() }`.

```

s := []string{"A", "B", "C", "D", "E", "F", "G"}
t := []string{"K", "L", "M", "N"}
u := []string{"m", "n", "o", "p", "q", "r"}
s = append(s, "h", "i", "j") // Добавить отдельные значения
s = append(s, t...)          // Добавить все элементы из среза t
s = append(s, u[2:5]...)     // Добавить часть элементов из среза u
b := []byte{'U', 'V'}
letters := "wxy"
b = append(b, letters...)    // Добавить байты из строки в срез с байтами
fmt.Printf("%v\n%s\n", s, b)
[A B C D E F G h i j K L M N o p q]
UVwxy

```

Встроенная функция `append()` принимает срез и одно или более добавляемых значений и возвращает (возможно, новый) срез, включающий содержимое оригинального среза, плюс указанное значение или значения, находящиеся в последних элементах. Если оригинальный срез имеет достаточную емкость (то есть сумма его длины и количества новых элементов не превышает емкости), функция `append()` вставит новые значения в конец среза и вернет оригинальный срез с длиной, увеличенной на количество добавленных элементов. Если оригинальный срез имеет недостаточную емкость, функция `append()` создаст новый срез, скопирует в него элементы из оригинального среза, добавит в конец новые значения и вернет новый срез – именно поэтому необходимо присваивать значение, возвращаемое функцией `append()`, оригинальной переменной.

Иногда бывает необходимо вставлять элементы не в конец, а в начало или в середину среза. Ниже приводятся несколько примеров применения пользовательской функции `InsertStringSliceCopy()`, принимающей срез, куда вставляются элементы, срез, элементы которого требуется вставить, и номер позиции для вставки.

```

s := []string{"M", "N", "O", "P", "Q", "R"}
x := InsertStringSliceCopy(s, []string{"a", "b", "c"}, 0) // В начало
y := InsertStringSliceCopy(s, []string{"x", "y"}, 3)      // В середину
z := InsertStringSliceCopy(s, []string{"z"}, len(s))      // В конец
fmt.Printf("%v\n%v\n%v\n%v\n%v\n", s, x, y, z)
[M N O P Q R]
[a b c M N O P Q R]
[M N O x y P Q R]
[M N O P Q R z]

```

Функция `InsertStringSliceCopy()` создает новый срез (именно поэтому срез `s` остался неизменным к концу выполнения фрагмента), вызывает встроенную функцию `copy()`, чтобы скопировать содержимое первого среза, и вставляет содержимое второго среза.

```
func InsertStringSliceCopy(slice, insertion []string, index int) []string {  
    result := make([]string, len(slice)+len(insertion))  
    at := copy(result, slice[:index])  
    at += copy(result[at:], insertion)  
    copy(result[at:], slice[index:])  
    return result  
}
```

Встроенная функция `copy()` принимает два среза (которые могут быть фрагментами одного и того же среза, причем даже перекрывающимися фрагментами), содержащие элементы одного типа. Она копирует элементы из второго (исходного) среза в первый (целевой) срез и возвращает количество скопированных элементов. Если исходный срез пуст, функция `copy()` ничего не сделает. Если длины целевого среза будет недостаточно, чтобы принять элементы из исходного среза, непоместившиеся элементы просто будут игнорироваться. Если емкость целевого среза больше его текущей длины, перед копированием длину можно увеличить до полной емкости с помощью инструкции `срез = срез[:cap(срез)]`.

Срезы, передаваемые встроенной функции `copy()`, должны иметь одинаковый тип. Исключение составляет случай, когда первый (целевой) срез имеет тип `[]byte`. В этой ситуации второй (исходный) аргумент может иметь тип `[]byte` *или* `string`. Если второй аргумент имеет тип `string`, функция просто скопирует байты строки в первый аргумент. (Пример использования этой возможности приводится в главе 6.)

Функция `InsertStringSliceCopy()` начинается с создания нового среза (`result`), достаточно большого, чтобы хранить элементы из обоих срезов, переданных функции. Затем в срез `result` копируется фрагмент первого среза (`slice[:index]`). Потом в срез `result`, начиная с позиции `at`, достигнутой в предыдущей операции копирования, копируется все содержимое среза `insertion`. После чего копируется остальное содержимое первого среза (`slice[index:]`). Результат последнего вызова функции `copy()` игнорируется, потому что он не нужен. И наконец, срез `result` возвращается вызывающей программе.

Если в аргументе `index` передать 0, выражение `slice[:index]` в первом вызове функции `copy()` примет вид `slice[:0]` (то есть пустой срез), поэтому ничего копироваться не будет. Аналогично если в аргументе `index` передать значение, большее или равное длине среза `slice`, выражение `slice[index:]` в последнем вызове функции `copy()` фактически примет вид `slice[len(slice):]` (то есть пустой срез), поэтому и в этой ситуации ничего не копируется.

Ниже приводится функция, обладающая практически тем же поведением, что и функция `InsertStringSliceCopy()`, но более коротка и простая. Разница в том, что `InsertStringSlice()` изменяет оригинальный срез (и, возможно, вставляемый срез), тогда как `InsertStringSliceCopy()` этого не делает.

```
func InsertStringSlice(slice, insertion []string, index int) []string {  
    return append(slice[:index], append(insertion, slice[index:]...)...)  
}
```

Функция `InsertStringSlice()` добавляет фрагмент оригинального среза, начиная с позиции `index` и до конца, в конец среза `insertion`, а затем добавляет получившийся срез в конец оригинального среза, начиная с позиции `index`. Возвращается оригинальный срез со вставленным в него срезом `insertion`. (Напомню, что функция `append()` принимает срез и одно или более значений, поэтому для преобразования среза в отдельные значения необходимо использовать оператор многоточия — в данном примере это необходимо сделать дважды.)

Удаление элементов в начале и в конце среза может выполняться с применением стандартного синтаксиса извлечения срезов, но удаление элементов из середины требует дополнительных операций. Сначала рассмотрим приемы удаления элементов в начале, в конце и в середине среза на месте. Затем посмотрим, как получить копию среза с удаленными элементами, оставив оригинальный срез неизменным.

```
s := []string{"A", "B", "C", "D", "E", "F", "G"}  
s = s[2:] // Удалит s[:2] из начала  
fmt.Println(s)  
[C D E F G]
```

Удаление элементов в начале среза легко реализуется с помощью операции извлечения среза.

```
s := []string{"A", "B", "C", "D", "E", "F", "G"}
s = s[:4] // Удалит s[4:] в конце
fmt.Println(s)
[A B C D]
```

Удаление элементов в конце среза также выполняется с помощью простой операции извлечения среза, как и при удалении элементов в начале.

```
s := []string{"A", "B", "C", "D", "E", "F", "G"}
s = append(s[1], s[5:]...) // Удалит s[1:5] из середины
fmt.Println(s)
[A F G]
```

Получить элементы из середины среза не составляет никакого труда. Например, получить три элемента из середины среза `s` можно с помощью выражения `s[2:5]`. Но удалить элементы из середины среза не так просто. Здесь удаление выполняется с вызовом функции `append()`, добавляющей фрагмент среза `s`, следующий за фрагментом, который требуется удалить, в конец фрагмента, предшествующего удаляемому, и присваиванием получившегося среза обратно переменной `s`.

Очевидно, что удаление элементов вызовом функции `append()` и присваиванием обратно оригинальной переменной изменяет оригинальный срез. Ниже приводятся несколько примеров применения пользовательской функции `RemoveStringSliceCopy()`, возвращающей копию указанного среза, из которой удалены элементы с указанными позициями.

```
s := []string{"A", "B", "C", "D", "E", "F", "G"}
x := RemoveStringSliceCopy(s, 0, 2) // Удалит s[:2] из начала
y := RemoveStringSliceCopy(s, 1, 5) // Удалит s[1:5] из середины
z := RemoveStringSliceCopy(s, 4, len(s)) // Удалит s[4:] в конце
fmt.Printf("%v\n%v\n%v\n%v\n", s, x, y, z)
[A B C D E F G]
[C D E F G]
[A F G]
[A B C D]
```

Функция `RemoveStringSliceCopy()` копирует элементы, поэтому оригинальный срез остается нетронутым.

```
func RemoveStringSliceCopy(slice []string, start, end int) []string {
    result := make([]string, len(slice)-(end-start))
```

```
    at := copy(result, slice[:start])
    copy(result[at:], slice[end:])
    return result
}
```

Функция `RemoveStringSliceCopy()` начинается с создания нового среза (`result`), достаточно большого, чтобы можно было сохранить в нем требуемые элементы. Затем в него копируются элементы из начала исходного среза до позиции `start` (`slice[:start]`). Потом в срез `result`, начиная с достигнутой к этому моменту позиции `at`, копируются элементы из конца исходного среза, начиная с позиции `end` (`slice[end:]`). И в заключение срез `result` возвращается вызывающей программе.

Имеется также возможность создавать более простые версии функции `RemoveStringSlice()`, оперирующие самим срезом, а не его копией.

```
func RemoveStringSlice(slice []string, start, end int) []string {
    return append(slice[:start], slice[end:]...)
}
```

Эта функция является обобщением приема удаления элементов и середины среза с помощью встроенной функции `append()`, продемонстрированного выше. Она возвращает оригинальный срез, из которого удалены элементы, начиная с позиции `start` и заканчивая (но не включая) позицией `end`.

4.2.4. Сортировка и поиск по срезам

Пакет `sort` из стандартной библиотеки содержит функции для сортировки срезов с числами типа `int`, `float64` и строками, для проверки сортировки таких срезов и для поиска элементов в отсортированных срезах с использованием быстрого алгоритма двоичного поиска. В пакете также имеются универсальные функции `sort.Sort()` и `sort.Search()`, упрощающие работу с данными пользовательских типов. Эти функции перечислены в табл. 4.2.

Механизм сортировки чисел, используемый в языке Go, не таит никаких сюрпризов, как было показано в главе 2 (§2.4). Однако строки сортируются по значениям байт, представляющих их, о чем уже рассказывалось в предыдущей главе (§3.2). Это означает, например, что сортировка строк выполняется с учетом регистра символов. Ниже приводятся пара примеров сортировки строк и результаты, которые они производят.

```

files := []string{"Test.conf", "util.go", "Makefile", "misc.go", "main.go"}
fmt.Printf("Unsorted:          %q\n", files)
sort.Strings(files)           // Функция сортировки из стандартной библиотеки
fmt.Printf("Underlying bytes: %q\n", files)
SortFoldedStrings(files) // Пользовательская функция сортировки
fmt.Printf("Case insensitive: %q\n", files)
Unsorted:          ["Test.conf" "util.go" "Makefile" "misc.go" "main.go"]
Underlying bytes: ["Makefile" "Test.conf" "main.go" "misc.go" "util.go"]
Case insensitive: ["main.go" "Makefile" "misc.go" "Test.conf" "util.go"]

```

Таблица 4.2. Функции из пакета *sort*

Функция	Описание/результат
<code>sort.Float64s(fs)</code>	Сортирует срез <code>fs</code> типа <code>[]float64</code> в порядке возрастания
<code>sort.Float64sAreSorted(fs)</code>	Возвращает <code>true</code> , если срез <code>fs</code> типа <code>[]float64</code> отсортирован
<code>sort.Ints(is)</code>	Сортирует срез <code>is</code> типа <code>[]int</code> в порядке возрастания
<code>sort.IntsAreSorted(is)</code>	Возвращает <code>true</code> , если срез <code>is</code> типа <code>[]int</code> отсортирован
<code>sort.IsSorted(d)</code>	Возвращает <code>true</code> , если срез <code>d</code> типа <code>sort.Interface</code> отсортирован
<code>sort.Search(size, fn)</code>	Возвращает номер позиции в отсортированном срезе, в области длиной <code>size</code> , где функция <code>fn</code> с сигнатурой <code>func(int) bool</code> возвращает <code>true</code> (см. описание в тексте)
<code>sort.SearchFloat64s(fs, f)</code>	Возвращает номер позиции элемента <code>f</code> типа <code>float64</code> в отсортированном срезе <code>fs</code> типа <code>[]float64</code>
<code>sort.SearchInts(is, i)</code>	Возвращает номер позиции элемента <code>i</code> типа <code>int</code> в отсортированном срезе <code>is</code> типа <code>[]int</code>
<code>sort.SearchStrings(ss, s)</code>	Возвращает номер позиции элемента <code>s</code> типа <code>string</code> в отсортированном срезе <code>ss</code> типа <code>[]string</code>
<code>sort.Sort(d)</code>	Сортирует срез <code>d</code> типа <code>sort.Interface</code> (см. пояснения в тексте)
<code>sort.Strings(ss)</code>	Сортирует срез <code>ss</code> типа <code>[]string</code> в порядке возрастания
<code>sort.StringsAreSorted(ss)</code>	Возвращает <code>true</code> , если срез <code>ss</code> типа <code>[]string</code> отсортирован

Функция `sort.Strings()` из стандартной библиотеки принимает срез типа `[]string` и сортирует строки на месте, в порядке возрастания по значениям составляющих их байт. Если все строки закодированы с применением одного и того же способа отображения

символов в байты (например, все они были созданы текущей программой или другой программой на языке Go), строки будут отсортированы по кодовым пунктам. Пользовательская функция `SortFoldedStrings()`, представленная ниже, действует точно так же, за исключением того, что она сортирует строки без учета регистра символов, используя универсальную функцию `sort.Sort()`.

Функция `sort.Sort()` способна сортировать элементы любого типа, предоставляющего методы, объявляемые интерфейсом `sort.Interface`, то есть элементы, имеющие методы `Len()`, `Less()` и `Swap()` с требуемыми сигнатурами. В следующем примере объявляется пользовательский тип `FoldedStrings`, реализующий эти методы. Ниже приводится полная реализация функции `SortFoldedStrings()` типа `FoldedStrings` и поддерживаемых им методов.

```
func SortFoldedStrings(slice []string) {
    sort.Sort(FoldedStrings(slice))
}

type FoldedStrings []string

func (slice FoldedStrings) Len() int { return len(slice) }
func (slice FoldedStrings) Less(i, j int) bool {
    return strings.ToLower(slice[i]) < strings.ToLower(slice[j])
}
func (slice FoldedStrings) Swap(i, j int) {
    slice[i], slice[j] = slice[j], slice[i]
}
}
```

Функция `SortFoldedStrings()` просто вызывает функцию `sort.Sort()` из стандартной библиотеки, преобразуя указанный срез типа `[]string` в значение типа `FoldedStrings` с помощью стандартного синтаксиса преобразования. Вообще говоря, всякий раз, когда определяется пользовательский тип, основанный на встроенном типе, значение встроенного типа можно преобразовать в значение пользовательского типа, используя простой синтаксис преобразования. (Подробнее о пользовательских типах рассказывается в главе 6.)

Тип `FoldedStrings` предоставляет три метода, объявляемые интерфейсом `sort.Interface`. Все методы имеют очень простую реализацию. Нечувствительность к регистру символов в этом типе достигается за счет использования функции `strings.ToLower()` в методе `Less()`. (Если потребуется реализовать сортировку в порядке возрастания, достаточно просто заменить в методе `Less()` оператор `<` на оператор `>`.)

Функция `SortFoldedStrings()` действует совершенно адекватно при обработке строк из 7-битных ASCII-символов (на английском языке), но маловероятно, что она будет давать удовлетворительные результаты упорядочения строк с текстом на других языках, о чем уже говорилось в предыдущей главе (§3.2). Сортировка строк Юникода с текстом на других языках, отличных от английского, – весьма нетривиальная задача. Эта проблема детально рассматривается в документе «Unicode Collation Algorithm» (unicode.org/reports/tr10).

При необходимости отыскать в срезе определенное значение (если имеется) это легко можно сделать с помощью цикла `for ... range`.

```
files := []string{"Test.conf", "util.go", "Makefile", "misc.go", "main.go"}
target := "Makefile"
for i, file := range files {
    if file == target {
        fmt.Printf("found \"%s\" at files[%d]\n", file, i)
        break
    }
}
found "Makefile" at files[2]
```

Простой линейный поиск, как в данном случае, пригоден только при работе с несортированными данными и только для небольших срезов (до нескольких сотен элементов). Но для огромных срезов, особенно когда поиск требуется выполнить многократно, линейный алгоритм оказывается весьма неэффективным – для поиска элемента в среднем приходится сравнить с искомым значением половину элементов.

В языке Go имеется метод `sort.Search()`, реализующий алгоритм *поиска методом половинного деления*: для выполнения поиска он требует выполнить не более $\log_2(n)$ сравнений (где n – количество элементов). Если взглянуть с этой точки зрения, алгоритм *линейного поиска* по 1 000 000 элементов требует в среднем выполнить 500 000 сравнений и в худшем случае – 1 000 000 сравнений; а алгоритм *поиска методом половинного деления* потребует выполнить максимум 20 сравнений, даже в самом худшем случае.

```
sort.Strings(files)
fmt.Printf("%q\n", files)
i := sort.Search(1e6, files),
func(i int) bool { return files[i] >= target }}
```

```

if i < len(files) && files[i] == target {
    fmt.Printf("found \"%s\" at files[%d]\n", files[i], i)
}
["Makefile" "Test.conf" "main.go" "misc.go" "util.go"]
found "Makefile" at files[0]

```

Функция `sort.Search()` принимает два аргумента: длину среза для поиска и функцию, сравнивающую элемент *отсортированного* среза с искомым значением с помощью оператора `>=` — для срезов, отсортированных в порядке возрастания, и `<=` — в порядке убывания. Функция должна образовывать *замыкание*, то есть она должна быть определена в области видимости среза, где предстоит выполнить поиск, так как она должна захватывать срез как часть своего окружения. (Замыкания подробно рассматриваются в §5.6.3.) Функция `sort.Search()` возвращает значение типа `int`. Если это значение меньше длины среза *и* в данной позиции находится искомый элемент, можно сделать вывод, что искомый элемент найден.

Ниже приводится пример поиска строки в срезе типа `[]string`, отсортированном без учета регистра символов, где предполагается, что искомая строка содержит только символы нижнего регистра.

```

target := "makefile"
SortFoldedStrings(files)
fmt.Printf("%q\n", files)
caseInsensitiveCompare := func(i int) bool {
    return strings.ToLower(files[i]) >= target
}
i := sort.Search(len(files), caseInsensitiveCompare)
if i < len(files) && strings.EqualFold(files[i], target) {
    fmt.Printf("found \"%s\" at files[%d]\n", files[i], i)
}
["main.go" "Makefile" "misc.go" "Test.conf" "util.go"]
found "Makefile" at files[1]

```

Здесь функция сравнения определена за пределами вызова функции `sort.Search()`. Однако обратите внимание, что, как и в предыдущем примере, функция сравнения образует замыкание с областью видимости среза, в котором выполняется поиск. Сравнение можно было бы выполнить с помощью инструкции `strings.ToLower(files[i]) == target`, но здесь используется более удобная функция `strings.EqualFold()`, сравнивающая две строки без учета регистра символов.

Срезы в языке Go являются настолько мощными, гибкими и удобными структурами данных, что сложно представить нетривиальную программу на языке Go, где им не нашлось бы места. Срезы еще будут встречаться далее в этой главе (§4.4).

Срезы являются, пожалуй, наиболее часто используемой структурой данных, тем не менее в некоторых ситуациях необходимо иметь возможность хранить данные в виде коллекций пар *ключ/значение*, обеспечивающих быстрый поиск по ключу. Данную возможность в языке Go обеспечивает тип `map` – тема следующего раздела.

4.3. Отображения

Отображения в языке Go – это неупорядоченные коллекции пар *ключ/значение*, емкость которых ограничивается лишь объемом доступной памяти¹. Ключи имеют уникальные значения в пределах коллекции и могут быть только одного типа, поддерживающего операторы `==` и `!=`, то есть в качестве ключей могут использоваться значения большинства встроенных типов (таких как `int`, `float64`, `rune`, `string`, сравнимые массивы и структуры, а также пользовательские типы, основанные на них, и указатели). Срезы, несравнимые массивы и структуры (элементов и полей которых не поддерживают операторы `==` и `!=`), а также пользовательские типы, основанные на них, не могут играть роль ключей в отображениях. В качестве значений могут использоваться указатели, значения ссылочных типов, а также значения встроенных или пользовательских типов, включая отображения, что позволяет легко создавать структуры данных произвольной сложности. Операции, поддерживаемые типом `map`, перечислены в табл. 4.3.

Отображения – это ссылочный тип, недорогой в передаче между функциями (всего 8 байт на 64-битных архитектурах и 4 байта – на 32-битных), независимо от объема данных, хранимых в отображениях. Поиск по отображениям выполняется очень быстро – намного быстрее линейного поиска, но примерно на два порядка (то есть в 100 раз) медленнее операции доступа к элементам массивов и срезов по их индексам, согласно неофициальным экспериментам². Тем не менее скорость поиска достаточно высока, чтобы можно было

¹ Отображения иногда называют хешами, хеш-таблицами, неупорядоченными отображениями, словарями и ассоциативными массивами.

² На момент написания этих строк не было доступно никакой официальной информации о времени поиска данных в отображениях.

использовать отображения, где это имеет смысл, поскольку на практике производительность редко оказывается большой проблемой. На рис. 4.5 схематически представлено строение отображения типа `map[string]float64`.

Поскольку срезы не могут играть роль ключей, может показаться, что в качестве ключей нельзя также использовать и срезы с байтами (`[]byte`). Однако, поскольку преобразования `string([]byte)` и `[]byte(string)` не изменяют байты данных, можно спокойно преобразовывать значения типа `[]byte` в тип `string`, использовать их как ключи отображения и преобразовывать обратно в значения типа `[]byte` при необходимости.

Все ключи отображения должны быть одного типа, то же относится и к значениям, хотя значения и ключи могут быть разных типов (как часто и бывает). Однако что касается значений в отображениях, на них, как и на элементы среза, практически не накладывается серьезных ограничений, потому что типом значения может быть интерфейс. То есть в отображении можно хранить значения любых типов, при условии что все они будут соответствовать указанному интерфейсу (иметь метод или методы, объявленные интерфейсом).

Таблица 4.3. Операции, поддерживаемые отображениями

Операция	Описание/результат
<code>m[k] = v</code>	Присвоит значение <code>v</code> ключу <code>k</code> в отображении <code>m</code> ; если ключ <code>k</code> уже имеется в отображении, его предыдущее значение будет затерто новым
<code>delete(m, k)</code>	Удалит ключ <code>k</code> и связанное с ним значение из отображения <code>m</code> ; если ключ <code>k</code> отсутствует в отображении – ничего не сделает
<code>v := m[k]</code>	Извлечет значение, соответствующее ключу <code>k</code> в отображении <code>m</code> , и присвоит его переменной <code>v</code> ; если ключ <code>k</code> отсутствует в отображении – присвоит нулевое значение соответствующего типа
<code>v, found := m[k]</code>	Извлечет значение, соответствующее ключу <code>k</code> в отображении <code>m</code> , и присвоит его переменной <code>v</code> , а переменной <code>found</code> присвоит значение <code>true</code> ; если ключ <code>k</code> отсутствует в отображении – присвоит переменной <code>v</code> нулевое значение соответствующего типа, а переменной <code>found</code> – значение <code>false</code>
<code>len(m)</code>	Количество элементов (пар <i>ключ/значение</i>) в отображении

Можно даже создать отображение со значениями, имеющими тип пустого интерфейса, `interface{}`, и хранить в нем значения любых типов, однако при обращении к ним придется использовать инструкции приведения типа, выбора по типу или механизм интроспекции, чтобы обеспечить корректное их использование.

(Интерфейсы подробно рассматриваются в главе 6; механизм рефлексии – в §9.4.9.)

Отображения создаются следующими способами:

```
make(map[ТипКлюча]ТипЗначения, начальнаяЕмкость)
make(map[ТипКлюча]ТипЗначения)
map[ТипКлюча]ТипЗначения {}
map[ТипКлюча]ТипЗначения{ключ1: значение1, ключ2: значение2, ..., ключN: значениеN}
```

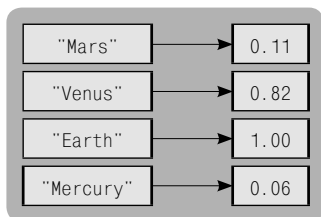


Рис. 4.5. Устройство отображения с ключами – строками и значениями типа `float64`

Встроенная функция `make()` используется для создания срезов, отображений и каналов. Когда она применяется для создания отображения, то создает пустое отображение, и если указано необязательное значение `начальнаяЕмкость`, для отображения выделяется объем памяти, достаточный для хранения указанного количества элементов. При попытке добавить в отображение больше элементов, чем позволяет начальная емкость, отображение будет

автоматически расширено. Второй и третий способы полностью эквивалентны. Два последних способа демонстрируют возможность создания отображений с применением синтаксиса *составных литералов* – это очень удобные и практичные способы создания пустых отображений или отображений с несколькими начальными значениями.

4.3.1. Создание и заполнение отображений

Следующие примеры демонстрируют создание и заполнение отображений с ключами типа `string` и значениями типа `float64`.

```
massForPlanet := make(map[string]float64) // То же, что и: map[string]float64{}
massForPlanet["Mercury"] = 0.06
massForPlanet["Venus"] = 0.82
massForPlanet["Earth"] = 1.00
massForPlanet["Mars"] = 0.11
fmt.Println(massForPlanet)
map[Venus:0.82 Mars:0.11 Earth:1 Mercury:0.06]
```

Для маленьких отображений совершенно не важно, будет ли определяться их начальная емкость, но для больших отображений это поможет повысить производительность. В общем случае начальную емкость желательно указывать, если она известна (хотя бы приблизительно).

Отображения поддерживают оператор индексирования `[]`, подобно массивам и срезам, только для отображений индексами внутри квадратных скобок служат ключи, которые могут быть не только целыми числами, но и, например, строками, как здесь показано.

Для вывода отображения в консоль здесь использована функция `fmt.Println()`. Она применяет спецификатор формата `%v` и выводит элементы отображения через пробел в форме *ключ: значение*. Отображения являются неупорядоченными коллекциями, поэтому на разных машинах порядок вывода элементов может отличаться от показанного здесь.

Как отмечалось выше, в качестве ключей отображения можно использовать указатели. Далее представлен пример использования подобных ключей типа `*Point`, где тип `Point` определен, как показано ниже:

```
type Point struct{ x, y, z int }
func (point Point) String() string {
    return fmt.Sprintf("(%d,%d,%d)", point.x, point.y, point.z)
}
```

Тип `Point` хранит три целых числа. Он имеет метод `String()`, обеспечивающий вывод значений полей стандартными функциями вывода в языке Go вместо простого адреса структуры `Point` в памяти.

При необходимости вывод адреса в памяти всегда можно обеспечить с помощью спецификатора формата `%p`. (О спецификаторах формата рассказывается выше, в §3.5.6.)

```
triangle := make(map[*Point]string, 3)
triangle[&Point{89, 47, 27}] = "α"
triangle[&Point{86, 65, 86}] = "β"
triangle[&Point{7, 44, 45}] = "γ"
fmt.Println(triangle)
map[(7,44,45):γ (89,47,27):β (86,65,86):β]
```

Здесь создается отображение с указанной начальной емкостью, которое заполняется ключами-указателями и строковыми значениями.

Каждое значение типа `Point` создается с использованием синтаксиса составных литералов, к которому тут же применяется оператор `&`, чтобы вместо значения типа `Point` получить значение типа `*Point`. (Этот прием был представлен ранее в этой главе, в §4.1.) И благодаря методу `Point.String()` при выводе отображения мы видим значения типа `*Point` в удобной для восприятия форме.

Использование *указателей* в качестве ключей отображения означает возможность добавить два значения типа `Point` с одинаковыми координатами, если они будут созданы независимо друг от друга (и, соответственно, будут иметь разные адреса в памяти). Но как быть, если потребуется хранить в отображении не более одной точки с конкретным набором координат? Этого легко можно добиться, используя в качестве ключей не указатели, а сами значения типа `Point` — в конце концов, в языке Go допускается использовать структуры в качестве ключей отображений, при условии что типы их полей поддерживают операции сравнения с помощью операторов `==` и `!=`. Например:

```
nameForPoint := make(map[Point]string) // То же, что и: map[Point]string{}
nameForPoint[Point{54, 91, 78}] = "x"
nameForPoint[Point{54, 158, 89}] = "y"
fmt.Println(nameForPoint)
map[{54,91,78}:x {54,158,89}:y]
```

Ключами в отображении `nameForPoint` являются уникальные значения типа `Point`, связанные со строками имен, которые можно изменить в любой момент времени.

```
populationForCity := map[string]int{"Istanbul": 12610000,
    "Karachi": 10620000, "Mumbai": 12690000, "Shanghai": 13680000}
for city, population := range populationForCity {
    fmt.Printf("%-10s %8d\n", city, population)
}
Shanghai 13680000
Mumbai 12690000
Istanbul 12610000
Karachi 10620000
```

В этом заключительном примере в данном подразделе отображение целиком было создано применением синтаксиса составных литералов.

Когда инструкция цикла `for ...range` применяется к отображению и указаны две переменные цикла, в каждой итерации она будет возвращать ключ и соответствующее ему значение, пока не выполнит обхода всех пар *ключ/значение*. Если в инструкции указана лишь одна переменная цикла, в каждой итерации в ней будет возвращаться очередной ключ. Поскольку отображения являются неупорядоченными коллекциями, заранее нельзя предсказать, в какой последовательности будут возвращаться элементы. В большинстве ситуаций требуется лишь обойти все элементы отображения, чтобы получить или изменить их значения, поэтому порядок итераций не имеет значения. Однако если потребуется выполнить итерации, например по упорядоченным ключам, это легко можно сделать, как будет показано ниже (§4.3.4).

4.3.2. Поиск в отображениях

В языке Go имеются две очень похожие операции *поиска в отображениях*, в каждой из которых используется оператор индексирования `[]`. Ниже приводится пара примеров самого простого способа.

```
population := populationForCity["Mumbai"]
fmt.Println("Mumbai's population is", population)
population = populationForCity["Emerald City"]
fmt.Println("Emerald City's population is", population)
Mumbai's population is 12690000
Emerald City's population is 0
```

Если выполняется поиск по ключу, присутствующему в отображении, возвращается соответствующее ему значение. Но если искомый ключ отсутствует, возвращается нулевое значение данного типа. То есть в примере выше по значению `0` для ключа `"Emerald City"` нельзя сказать, означает ли оно, что в городе `Emerald City` действительно отсутствует население или этот город отсутствует в отображении. Решение этой проблемы предлагает второй способ поиска.

```
city := "Istanbul"
if population, found := populationForCity[city]; found {
    fmt.Printf("%s's population is %d\n", city, population)
} else {
    fmt.Printf("%s's population data is unavailable\n", city)
}
city = "Emerald City"
```

```
_, present := populationForCity[city]
fmt.Printf("%q is in the map == %t\n", city, present)
Istanbul's population is 12610000
"Emerald City" is in the map == false
```

При использовании *двух* переменных в операции индексирования отображений первой из них присваивается значение соответствующего ключа (или нулевое значение, если искомый ключ отсутствует в отображении), а второй – значение `true` (или `false`, если искомый ключ отсутствует). Это позволяет проверить наличие ключа в отображении. И как показывает второй пример, можно использовать пустой идентификатор, чтобы отбросить значение, если требуется всего лишь проверить наличие определенного ключа в отображении.

4.3.3. Изменение отображений

Элементы, являющиеся парами *ключ/значение*, можно вставлять в отображения и удалять из них. Кроме того, любое значение, связанное с тем или иным ключом, можно изменить в любой момент. Ниже приводятся несколько показательных примеров.

```
fmt.Println(len(populationForCity), populationForCity)
delete(populationForCity, "Shanghai") // Удаление
fmt.Println(len(populationForCity), populationForCity)
populationForCity["Karachi"] = 11620000 // Изменение
fmt.Println(len(populationForCity), populationForCity)
populationForCity["Beijing"] = 11290000 // Вставка
fmt.Println(len(populationForCity), populationForCity)
4 map[Shanghai:13680000 Mumbai:12690000 Istanbul:12610000 Karachi:10620000]
3 map[Mumbai:12690000 Istanbul:12610000 Karachi:10620000]
3 map[Mumbai:12690000 Istanbul:12610000 Karachi:11620000]
4 map[Mumbai:12690000 Istanbul:12610000 Karachi:11620000 Beijing:11290000]
```

Операции вставки и изменения элементов отображений имеют идентичный синтаксис: если элемент с указанным ключом отсутствует, вставляется новый элемент с данным ключом и значением; если элемент с указанным ключом присутствует, соответствующее ему значение замещается указанным, а оригинальное значение утрачивается. Если попытаться удалить элемент, отсутствующий в отображении, просто ничего не произойдет.

Сами ключи не могут изменяться, однако добиться эффекта изменения ключа можно следующим образом:

```
oldKey, newKey := "Beijing", "Tokyo"
value := populationForCity[oldKey]
delete(populationForCity, oldKey)
populationForCity[newKey] = value
fmt.Println(len(populationForCity), populationForCity)
4 map[Mumbai:12690000 Istanbul:12610000 Karachi:11620000 Tokyo:11290000]
```

Здесь сначала извлекается значение ключа, подлежащего изменению, затем из отображения удаляется элемент с этим ключом и создается другой элемент с новым ключом и прежним значением.

4.3.4. Итерации по отображениям с упорядоченными ключами

При воспроизведении данных для представления человеку часто бывает необходимо расположить их в определенном порядке. Следующий пример демонстрирует вывод содержимого отображения `populationForCity`, упорядоченного по именам городов в алфавитном порядке (строго говоря – в порядке величин кодовых пунктов Юникода).

```
cities := make([]string, 0, len(populationForCity))
for city := range populationForCity {
    cities = append(cities, city)
}
sort.Strings(cities)
for _, city := range cities {
    fmt.Printf("%-10s %8d\n", city, populationForCity[city])
}
Beijing 11290000
Istanbul 12610000
Karachi 11620000
Mumbai 12690000
```

Здесь сначала создается срез типа `[]string` нулевой длины (то есть пустой), но с достаточной емкостью, чтобы сохранить в нем все ключи отображения. Затем выполняются итерации по отображению, в ходе которых извлекаются только ключи (поскольку используется лишь одна переменная цикла, `city`, а не две, необходимые для извлечения пар *ключ/значение*) и добавляются в срез `cities`. Далее выполняется сортировка среза, затем выполняются итерации по срезу (игнорируя целочисленный индекс с помощью пустого

идентификатора) и отыскиваются соответствующие значения численности населения.

Представленный здесь алгоритм – создание пустого среза достаточной емкости, добавление в срез всех ключей отображения, сортировка среза и выполнение итераций по нему, чтобы обеспечить вывод упорядоченных данных, – является типичным алгоритмом итераций по отображениям с упорядоченными ключами.

Альтернативный способ заключается в использовании изначально упорядоченной структуры данных, например упорядоченного отображения. Пример такой структуры данных будет представлен в главе 6 (§6.5.3).

Имеется также возможность упорядочить вывод по значениям, например за счет инвертирования отображения, как будет показано в следующем подразделе.

4.3.5. Инвертирование отображений

Отображения, значения в которых являются уникальными и тип которых позволяет использовать их в качестве ключей, легко можно *инвертировать*. Например:

```
cityForPopulation := make(map[int]string, len(populationForCity))
for city, population := range populationForCity {
    cityForPopulation[population] = city
}
fmt.Println(cityForPopulation)
map[12610000:Istanbul 11290000:Beijing 12690000:Mumbai 11620000:Karachi]
```

Этот пример начинается с создания инвертированного отображения, то есть с преобразования отображения `populationForCity`, имеющего тип `map[string]int`, в отображение `cityForPopulation` с типом `map[int]string`. Затем выполняются итерации по оригинальному отображению, и его элементы вставляются в инвертированное отображение, меняя местами ключи и значения.

Разумеется, операция инвертирования будет терпеть неудачу, если значения не уникальны – по сути, неудача заключается в том, что в неуникальном ключе инвертированного отображения будет сохранено последнее встретившееся значение. Эту проблему можно решить за счет создания инвертированного отображения, позволяющего сохранять несколько значений для каждого ключа. Так, в данном примере можно было бы создать инвертированное отображение

типа `map[int][]string` (целочисленные ключи и значения типа `[]string`). Практический пример использования такого подхода будет представлен чуть ниже (§4.4.2).

4.4. Примеры

В этом разделе представлены два маленьких примера. Первый иллюстрирует использование одно- и двумерных срезов, а второй – применение отображений, включая инвертирование отображений, где значения могут оказаться неуникальными, а также использование срезов и сортировку.

4.4.1. Пример: угадай разделитель

В некоторых ситуациях может потребоваться принять для обработки группу файлов, где в каждом файле имеются записи, по одной на строке, но в различных файлах могут использоваться различные разделители полей (например, табуляции, пробелы или символ «*»). Для обработки целых групп таких файлов необходимо иметь возможность определять разделитель, используемый в каждом файле. Пример `guess_separator`, представленный в этом разделе (в файле `file_guess_separator/guess_separator.go`), пытается идентифицировать разделитель, используемый в файле, переданном для обработки.

Ниже приводится пример типичного сеанса работы с программой:

```
$ ./guess_separator information.dat
tab-separated
```

Программа читает первые пять строк (или меньше, если файл содержит менее пяти строк) и по ним пытается определить, какой разделитель используется в этом файле.

Как обычно, рассмотрение примера начнется с функции `main()`, за которой будут рассматриваться функции, вызываемые из нее (кроме одной). Но мы пропустим раздел импортирования модулей.

```
func main() {
    if len(os.Args) == 1 || os.Args[1] == "-h" || os.Args[1] == "--help" {
        fmt.Printf("usage: %s file\n", filepath.Base(os.Args[0]))
        os.Exit(1)
    }
    separators := []string{"\t", "*", "|", "•"}
```

```

linesRead, lines := readUpToNLines(os.Args[1], 5)
counts := createCounts(lines, separators, linesRead)
separator := guessSep(counts, separators, linesRead)
report(separator)
}

```

Функция `main()` начинается с проверки наличия имени файла в аргументах командной строки, при его отсутствии функция выводит сообщение о порядке использования и завершает программу.

Затем создается срез типа `[]string` с возможными разделителями – для файлов, где роль разделителя играет пробел, будет считаться, что разделителем является пустая строка (`""`).

Первое настоящее действие – чтение пяти первых строк из файла. Функция `readUpToNLines()` здесь не показана, потому что ранее уже приводились примеры чтения строк из файлов (и еще один пример будет показан в следующем разделе). Единственная необычная особенность функции `readUpToNLines()` состоит в том, что она читает только указанное число строк, или меньше, если файл содержит меньшее число строк, и возвращает число фактически прочитанных строк вместе с самими строками.

Остальные функции будут рассматриваться в порядке их вызова из функции `main()`, начиная с функции `createCounts()`.

```

func createCounts(lines, separators []string, linesRead int) [][]int {
    counts := make([][]int, len(separators))
    for sepIndex := range separators {
        counts[sepIndex] = make([]int, linesRead)
        for lineIndex, line := range lines {
            counts[sepIndex][lineIndex] =
                strings.Count(line, separators[sepIndex])
        }
    }
    return counts
}

```

Назначение функции `createCounts()` – заполнить матрицу, хранящую счетчики для каждого разделителя в каждой прочитанной строке.

Начинается функция с создания срезов с целыми числами, где число срезов совпадает с числом возможных разделителей. Для случая с четырьмя разделителями будет создан срез `counts: [nil nil nil nil]`. Внешний цикл `for` замещает каждое значение `nil` срезом

типа `[]int`, число элементов в котором соответствует числу прочитанных строк. То есть каждое значение `nil` замещается срезом `[0 0 0 0 0]`, так как в языке Go значения всегда инициализируются нулевыми значениями соответствующего типа.

Вложенный цикл `for` заполняет срез `counts`. В каждой строке подсчитывается количество вхождений каждого разделителя и сохраняется в соответствующем элементе среза `counts`. Функция `strings.Count()` возвращает количество вхождений своего второго строкового аргумента в первом строковом аргументе.

Например, допустим, что имеется файл, где в качестве разделителя используется символ табуляции, в котором в некоторых полях встречаются пробелы, звездочки и маркеры, и для него была получена матрица `counts`: `[[3 3 3 3 3] [0 0 4 3 0] [0 0 0 0 0] [1 2 2 0 0]]`. Каждый элемент среза `counts` является срезом типа `[]int`, содержащим счетчики для соответствующих разделителей (табуляция, звездочка, вертикальная черта, маркер) для каждой из пяти строк. В данном случае каждая строка содержит по три символа табуляции, пара строк содержит звездочки (четыре в одной и три в другой), три строки содержат маркеры, и ни в одной строке не встречается символа вертикальной черты. Для человека уже очевидно, что здесь в качестве разделителя используется символ табуляции, но программе еще предстоит установить этот факт, для чего используется функция `guessSep()`.

```
func guessSep(counts [][]int, separators []string, linesRead int) string {
    for sepIndex := range separators {
        same := true
        count := counts[sepIndex][0]
        for lineIndex := 1; lineIndex < linesRead; lineIndex++ {
            if counts[sepIndex][lineIndex] != count {
                same = false
                break
            }
        }
        if count > 0 && same {
            return separators[sepIndex]
        }
    }
    return ""
}
```

Цель этой функции – отыскать первый срез `[]int` в срезе `counts`, содержащий одинаковые, ненулевые значения во всех элементах.

Функция выполняет итерации по всем «строкам» в срезе `counts` (по одной на каждый разделитель) и первоначально предполагает, что все значения в строке одинаковы. Она инициализирует переменную `count` значением первого счетчика, то есть количеством вхождений данного разделителя в первой прочитанной строке. Затем выполняет итерации по остальным значениям, то есть количествам вхождений разделителя в других прочитанных строках. При обнаружении отличающегося значения внутренний цикл `for` прерывается, и проверяется следующий разделитель. Если внутренний цикл `for` заканчивается, так и не присвоив переменной `same` значение `false`, и значение `count` оказывается больше нуля, делается предположение, что искомый разделитель найден, и он немедленно возвращается вызывающей программе. Если ни один из ожидаемых разделителей не соответствует установленным критериям, возвращается пустая строка – в соответствии с принятыми здесь соглашениями это означает, что поля отделяются пробелами или вообще не отделяются никак.

```
func report(separator string) {
    switch separator {
    case "":
        fmt.Println("whitespace-separated or not separated at all")
    case "\t":
        fmt.Println("tab-separated")
    default:
        fmt.Printf("%s-separated\n", separator)
    }
}
```

Функция `report()` просто выводит описание разделителя, найденного в прочитанном файле.

Этот пример демонстрирует типичное использование одно- и двумерных срезов (разделители, строки и счетчики). Следующий пример демонстрирует использование отображений, срезов и реализацию сортировки.

4.4.2. Пример: частота встречаемости слов

Текстовый анализ имеет самые разные применения, от сбора данных до лингвистических исследований. В этом разделе рассматривается

пример, реализующий одну из простейших форм текстового анализа: он подсчитывает частоту встречаемости слов в указанных файлах.

Частотные данные можно представить двумя разными, но одинаково значимыми способами – в виде алфавитного списка слов с частотой их встречаемости, и список, упорядоченный по частоте встречаемости, где каждой частоте соответствует список слов. Программа `wordfrequency` (в файле `wordfrequency/wordfrequency.go`) выводит оба списка, как показано ниже.

```
$ ./wordfrequency small-file.txt
Word      Frequency
ability    1
about      1
above      3
...
years      1
you        128
Frequency → Words
  1 ability, about, absence, absolute, absolutely, abuse, accessible, ...
  2 accept, acquired, after, against, applies, arrange, assumptions, ...
...
128 you
151 or
192 to
221 of
345 the
```

Даже для небольшого файла количество разных слов и частот может оказаться достаточно большим, поэтому здесь большая часть вывода опущена.

Реализация вывода первого списка выглядит довольно просто. Для этого достаточно использовать отображение типа `map[string]int`, где роль ключей играют слова, а значений – частота встречаемости. Но, чтобы вывести второй список, необходимо инвертировать отображение, а это не так просто, потому что весьма вероятно наличие в файле нескольких слов с одинаковой частотой встречаемости. Решение заключается в инвертировании исходного отображения в отображение типа `map[int][]string`, где роль ключей играют значения частоты встречаемости, а роль значений – списки соответствующих им слов.

Начнем исследование программы с функции `main()` и затем будем двигаться сверху вниз, как обычно, опустив раздел импортирования модулей.

```

func main() {
    if len(os.Args) == 1 || os.Args[1] == "-h" || os.Args[1] == "--help" {
        fmt.Printf("usage: %s <file1> [<file2> [... <fileN>]]\n",
            filepath.Base(os.Args[0]))
        os.Exit(1)
    }
    frequencyForWord := map[string]int{} // То же, что и: make(map[string]int)
    for _, filename := range commandLineFiles(os.Args[1:]) {
        updateFrequencies(filename, frequencyForWord)
    }
    reportByWords(frequencyForWord)
    wordsForFrequency := invertStringIntMap(frequencyForWord)
    reportByFrequency(wordsForFrequency)
}

```

Функция `main()` начинается с обработки аргументов командой строки и затем приступает к работе.

Работа начинается с создания простого отображения для хранения частот встречаемости слов, прочитанных из файла. Изначально пустое отображение создается с помощью синтаксиса составных литералов, просто чтобы показать, как это делается. После создания отображения выполняются итерации по именам файлов, полученным из командной строки, и для каждого из них выполняется попытка дополнить отображение `frequencyForWord`.

После заполнения отображения выводится первый отчет: алфавитный список слов с соответствующими им частотами встречаемости. Затем создается инвертированная версия отображения и выводится второй отчет: список частот в числовом порядке с соответствующими им словами.

```

func commandLineFiles(files []string) []string {
    if runtime.GOOS == "windows" {
        args := make([]string, 0, len(files))
        for _, name := range files {
            if matches, err := filepath.Glob(name); err != nil {
                args = append(args, name) // Недопустимый шаблон имени
            } else if matches != nil {    // Как минимум одно совпадение
                args = append(args, matches...)
            }
        }
        return args
    }
    return files
}

```

На Unix-подобных платформах, таких как Linux и Mac OS X, функция `commandLineFiles()` просто возвращает срез типа `[]string` в том виде, в каком она его получила, потому что на этих платформах командная оболочка автоматически выполняет *подстановку имен файлов* (globbing) (то есть замещает шаблоны, такие как `*.txt`, именами соответствующих им файлов, например `README.txt`, `INSTALL.txt` и др.). Командная оболочка в Windows (`cmd.exe`) не выполняет подстановки имен файлов, поэтому если пользователь передаст в командной строке, например, шаблон `*.txt`, именно в таком виде он и попадет в программу. Чтобы обеспечить совместимость с различными платформами, необходимо реализовать подстановку имен файлов для ситуации, когда программа выполняется в Windows. (Другой способ обеспечения совместимости с разными платформами состоит в том, чтобы создавать файлы `.go` для каждой разновидности платформ – эта тема рассматривается в последней главе, в §9.1.1.1.)

```
func updateFrequencies(filename string, frequencyForWord map[string]int) {
    var file *os.File
    var err error
    if file, err = os.Open(filename); err != nil {
        log.Println("failed to open the file: ", err)
        return
    }
    defer file.Close()
    readAndUpdateFrequencies(bufio.NewReader(file), frequencyForWord)
}
```

Данная функция используется исключительно для обработки файлов. Она открывает указанный файл на чтение, откладывает закрытие файла до своего завершения и перекладывает всю фактическую работу на функцию `readAndUpdateFrequencies()`. Передача значения типа `*bufio.Reader` (создается вызовом `bufio.NewReader()`) обеспечивает возможность чтения содержимого файла в виде строк, а не последовательностей байт.

```
func readAndUpdateFrequencies(reader *bufio.Reader,
    frequencyForWord map[string]int) {
    for {
        line, err := reader.ReadString('\n')
        for _, word := range SplitOnNonLetters(strings.TrimSpace(line)) {
```

```
    if len(word) > utf8.UTFMax ||  
        utf8.RuneCountInString(word) > 1 {  
        frequencyForWord[strings.ToLower(word)] += 1  
    }  
}  
if err != nil {  
    if err != io.EOF {  
        log.Println("failed to finish reading the file: ", err)  
    }  
    break  
}  
}
```

Первая часть функции должна показаться знакомой читателю. Здесь определяется бесконечный цикл, в каждой итерации которого выполняется чтение строки из файла, прерывающийся по достижении конца файла или появления ошибки (в этом случае выводится сообщение об ошибке). При появлении ошибки программа не завершается, потому что ей может быть передано несколько имен файлов, и было бы предпочтительнее выполнить как можно больше работы, сообщая о возникших проблемах, вместо того чтобы остановиться на первой же встретившейся ошибке.

Все самое интересное происходит во внутреннем цикле `for`. В любой строке могут встретиться знаки пунктуации, цифры и другие символы, которые не могут входить в состав слов. Поэтому здесь вызовом функции `SplitOnNonLetters()` строка разбивается на слова по символам, не являющимся алфавитными, и затем выполняются итерации по полученным словам. Но, прежде чем передать этой функции строку из файла, из нее предварительно удаляются начальные и завершающие пробельные символы.

Единственное ограничение состоит в том, что включаться в отображение должны слова, содержащие не меньше двух символов. Это ограничение легко можно было бы наложить простой инструкцией `if`, проверяющей единственное условие, то есть `if utf8.RuneCountInString(word) > 1`, прекрасно справляющейся со своей задачей.

Но упомянутая простая инструкция `if` является достаточно дорогостоящей, потому что она выполняет разбор всего слова. Поэтому в программе используется инструкция `if` с двумя условиями, первое из которых проверяется очень быстро. Первое условие сравнивает количество байтов в слове со значением константы `utf8.UTFMax`

(которое равно 4 – максимальному числу байт, необходимому для представления единственного символа в кодировке UTF-8). Эта проверка действительно выполняется очень быстро, потому что в языке Go для строк известно, сколько байт они содержат, а выражения с двухместными логическими операторами (&& и ||) вычисляются по короткой схеме (§2.2). Разумеется, слова, состоящие из четырех байт или меньше (например, из четырех 7-битных ASCII-символов или из двух 2-байтных символов UTF-8), не будут проходить эту проверку. Но это не проблема, потому что вторая проверка (выполняющая подсчет символов) также будет выполняться быстро, из-за того что она всегда будет проверять слова, состоящие из четырех символов или меньше. Оправдано ли в данной ситуации использовать инструкцию `if` с двумя условиями? В действительности это во многом зависит от исходных данных – чем больше слов в файлах и чем длиннее эти слова, тем больше будет выигрыш в производительности. Единственный способ оценить этот выигрыш – провести тестирование с использованием настоящих или хотя бы типичных данных.

```
func SplitOnNonLetters(s string) []string {
    notALetter := func(char rune) bool { return !unicode.IsLetter(char) }
    return strings.FieldsFunc(s, notALetter)
}
```

Эта функция разбивает строку по неалфавитным символам. Сначала в ней создается анонимная функция с сигнатурой, требуемой функцией `strings.FieldsFunc()`, которая возвращает `true` для неалфавитных символов и `false` – для букв. Затем вызывающей программе возвращается результат вызова функции `strings.FieldsFunc()`, которой передаются указанная строка и функция `notALetter()`. (Функция `strings.FieldsFunc()` обсуждалась в предыдущей главе, в §3.6.1.)

```
func reportByWords(frequencyForWord map[string]int) {
    words := make([]string, 0, len(frequencyForWord))
    wordWidth, frequencyWidth := 0, 0
    for word, frequency := range frequencyForWord {
        words = append(words, word)
        if width := utf8.RuneCountInString(word); width > wordWidth {
            wordWidth = width
        }
    }
```



```

    if width := len(fmt.Sprint(frequency)); width > frequencyWidth {
        frequencyWidth = width
    }
}
sort.Strings(words)
gap := wordWidth + frequencyWidth - len("Word") - len("Frequency")
fmt.Printf("Word %s%s\n", gap, " ", "Frequency")
for _, word := range words {
    fmt.Printf("%-*s %d\n", wordWidth, word, frequencyWidth,
        frequencyForWord[word])
}
}

```

После заполнения отображения `frequencyForWord` вызывается функция `reportByWords()` для вывода данных. Здесь требуется вывести слова в алфавитном порядке (фактически в порядке возрастания кодовых пунктов Юникода), поэтому для начала создается пустой срез `[]string` достаточной емкости, чтобы поместить в него все слова, имеющиеся в отображении. Здесь также необходимо определить ширину в символах самого длинного слова и самое большое значение частоты встречаемости (то есть количество цифр в этом значении), чтобы можно было отформатировать вывод по столбцам: для хранения этих данных используются переменные `wordWidth` и `frequencyWidth`.

Первый цикл `for` выполняет итерации по элементам отображения. Каждое слово добавляется в срез `words []string`, добавление происходит очень быстро, потому что срез имеет достаточную емкость, чтобы на долю функции `append()` оставалось лишь сохранять слово в элементе с индексом `len(words)` и увеличивать длину среза `words` на единицу.

Для каждого слова подсчитывается количество символов и, если оно больше значения переменной `wordWidth`, сохраняется в этой переменной. Аналогично подсчитывается количество символов, необходимое для отображения частоты встречаемости, — здесь можно смело использовать функцию `len()`, потому что функция `fmt.Sprint()` принимает число и возвращает строку из десятичных цифр, каждая из которых является 7-битным ASCII-символом. Таким образом, к концу первого цикла определяется ширина для каждой из двух колонок.

После заполнения среза `words` выполняется его сортировка. Здесь нет причин волноваться о чувствительности к регистру символов,

потому что все слова уже преобразованы к нижнему регистру (в функции `readAndUpdateFrequencies()`; выше).

После сортировки выводятся заголовки двух колонок. Сначала выводится заголовок «Word» (слово), затем пробелы, чтобы выровнять правый край заголовка «Frequency» (частота) по последней цифре в колонке со значениями частот. Для этого выводится единственный пробел (" ") в поле с шириной в `gap` символов с помощью спецификатора формата `%*s`. Точно так же можно было бы использовать обычный спецификатор `%s` и передать ему строку из пробелов, созданную вызовом функции `strings.Repeat(" ", gap)`. (Спецификаторы формата подробно рассматриваются в предыдущей главе, в §3.5.)

И в заключение выводятся слова с соответствующими значениями частоты встречаемости. Слова и значения частот выводятся в двух колонках соответствующей ширины в алфавитном порядке следования слов.

```
func invertStringIntMap(intForString map[string]int) map[int][]string {
    stringsForInt := make(map[int][]string, len(intForString))
    for key, value := range intForString {
        stringsForInt[value] = append(stringsForInt[value], key)
    }
    return stringsForInt
}
```

Функция начинается с создания пустого *инвертированного отображения*. И хотя заранее неизвестно, сколько элементов в нем будет, можно смело предположить, что их будет не больше, чем элементов в оригинальном отображении. Само инвертирование выполняется достаточно просто: функция просто выполняет итерации по оригинальному отображению и, используя каждое значение как ключ в инвертированном отображении, добавляет каждый ключ оригинального отображения в соответствующий срез-значение в инвертированном отображении. Поскольку значениями в новом отображении являются срезы, никакие данные не будут утрачены, даже если в оригинальном срезе встретятся несколько ключей с одинаковыми значениями.

```
func reportByFrequency(wordsForFrequency map[int][]string) {
    frequencies := make([]int, 0, len(wordsForFrequency))
    for frequency := range wordsForFrequency {
        frequencies = append(frequencies, frequency)
    }
}
```

```

sort.Ints(frequencies)
width := len(fmt.Sprint(frequencies[len(frequencies)-1]))
fmt.Println("Frequency → Words")
for _, frequency := range frequencies {
    words := wordsForFrequency[frequency]
    sort.Strings(words)
    fmt.Printf("%*d %s\n", width, frequency, strings.Join(words, ", "))
}
}

```

Эта функция по своей структуре очень похожа на функцию `reportByWords()`. Она начинается с создания среза со значениями частот, который затем сортирует в порядке возрастания. Далее она вычисляет ширину поля для вывода самого большого значения частоты и использует его в качестве ширины первой колонки. Затем она выводит заголовок отчета. И наконец, выполняет итерации по элементам среза `frequencies` и выводит каждый из них со списком соответствующих ему слов в алфавитном порядке и через запятую, если их больше одного.

Теперь, закончив обзор двух примеров для этой главы, вы должны иметь некоторое представление об использовании указателей в языке Go, а также о возможностях срезов и отображений. В следующей главе будет рассказываться о создании пользовательских функций – это будет законченное введение в основы процедурного программирования на языке Go. После близкого знакомства с функциями можно будет перейти к обсуждению темы объектно-ориентированного, а затем и параллельного программирования.

4.5. Упражнения

Здесь предлагается выполнить пять упражнений, в каждом из которых требуется написать небольшую функцию и воспользоваться знаниями о срезах и отображениях, полученными в этой главе. В примере решения все пять функций помещены в один файл (`chap4_ans/chap4_ans.go`), куда также была добавлена функция `main()`, чтобы упростить тестирование. (О модульном тестировании рассказывается в главе 9, в §9.1.1.3.)

1. Создайте функцию, принимающую срез `[]int` и возвращающую срез `[]int`, – копию исходного среза, из которой удалены повторяющиеся значения. Например, пусть имеется срез `[]int{9, 1, 9, 5, 4, 4, 2, 1, 5, 4, 8, 8, 4, 3, 6, 9, 5,`

7, 5}, функция должна вернуть срез `[]int{9, 1, 5, 4, 2, 8, 3, 6, 7}`. В файле с решением `chap4_ans.go` эта функция называется `UniqueInts()`. Вместо встроенной функции `make()` она использует составной литерал и имеет длину 11 строк. Решить это упражнение будет несложно.

2. Создайте функцию, принимающую срез `[][]int` (то есть двумерный срез с целыми числами) и возвращающую одномерный срез `[]int`, содержащий все целые числа из первого среза в двумерном срезе, затем из второго среза и т. д. Например, если принять, что требуемая функция называется `Flatten()`:

```
irregularMatrix := [][]int{{1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11},
    {12, 13, 14, 15},
    {16, 17, 18, 19, 20}}
slice := Flatten(irregularMatrix)
fmt.Printf("1x%d: %v\n", len(slice), slice)
1x20: [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20]
```

Функция `Flatten()` в файле `chap4_ans.go` занимает всего девять строк. Проверка правильности работы функции имеет свои тонкости, даже если длины внутренних срезов отличаются (как в примере матрицы `irregularMatrix`), но в целом выполняется достаточно просто.

3. Создайте функцию, принимающую срез `[]int` и количество столбцов (в виде значения типа `int`) и возвращающую срез `[][]int`, где каждый внутренний срез имеет длину, равную указанному числу столбцов. Например, пусть имеется срез `[]int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}`, тогда требуемая функция должна возвращать срезы, как показано ниже, где в начале каждой строки указано количество заданных столбцов:

```
3 [[1 2 3] [4 5 6] [7 8 9] [10 11 12] [13 14 15] [16 17 18] [19 20 0]]
4 [[1 2 3 4] [5 6 7 8] [9 10 11 12] [13 14 15 16] [17 18 19 20]]
5 [[1 2 3 4 5] [6 7 8 9 10] [11 12 13 14 15] [16 17 18 19 20]]
6 [[1 2 3 4 5 6] [7 8 9 10 11 12] [13 14 15 16 17 18] [19 20 0 0 0 0]]
```

Обратите внимание: поскольку число 20 (количество значений в исходном срезе) не делится без остатка на 3 и 6, последний внутренний срез был дополнен нулевыми значениями, чтобы обеспечить равное количество столбцов во всех срезах.

Функция `Make2D()` в файле `chap4_ans.go` занимает 12 строк и использует вспомогательную функцию, занимающую 7 строк. При создании функции `Make2D()` и ее вспомогательной функции придется немного подумать, как правильнее решить поставленную задачу, но в целом в них нет ничего сложного.

4. Создайте функцию, принимающую срез `[]string` со строками из `ini`-файла и возвращающую отображение `map[string]map[string]string`, где ключами являются имена групп, а значениями — отображения с ключами и значениями из этих групп. Пустые строки и строки, начинающиеся с символа `;`, должны игнорироваться. Каждая группа начинается строкой с именем группы в квадратных скобках, а ключи и значения в группе могут занимать одну или более строк и имеют вид *ключ=значение*. Ниже приводится пример содержимого среза `[]string`, который должна уметь обрабатывать функция.

```
iniData := []string{
    "; Cut down copy of Mozilla application.ini file",
    "",
    "[App]",
    "Vendor=Mozilla",
    "Name=Iceweasel",
    "Profile=mozilla/firefox",
    "Version=3.5.16",
    "[Gecko]",
    "MinVersion=1.9.1",
    "MaxVersion=1.9.1.*",
    "[XRE]",
    "EnableProfileMigrator=0",
    "EnableExtensionManager=1",
}
```

Для этих данных функция должна вернуть следующее отображение, в которое были добавлены отступы, чтобы проще было увидеть его структуру.

```
map[Gecko: map[MinVersion: 1.9.1
                MaxVersion: 1.9.1.*]]
```

```
XRE: map[EnableProfileMigrator: 0
        EnableExtensionManager: 1]
App: map[Vendor: Mozilla
        Profile: mozilla/firefox
        Name: Iceweasel
        Version: 3.5.16]]
```

Функция `ParseIni()` в предложенном решении помещает пары ключ/значение, определяемые за пределами каких-либо групп, в группу «General». Она занимает 24 строки, и для ее реализации придется хорошо подумать.

5. Создайте функцию, которая будет принимать отображение `map[string]map[string]string`, представляющее данные из ini-файла. Функция должна выводить эти данные в формате ini-файла с группами и ключами внутри групп, следующими в алфавитном порядке, и с пустыми строками между группами. Например, пусть имеются исходные данные, полученные в предыдущем упражнении, тогда функция должна производить следующий вывод:

```
[App]
Name=Iceweasel
Profile=mozilla/firefox
Vendor=Mozilla
Version=3.5.16
[Gecko]
MaxVersion=1.9.1.*
MinVersion=1.9.1
[XRE]
EnableExtensionManager=1
EnableProfileMigrator=0
```

Функция `PrintIni()` в предложенном решении занимает 21 строку и проще в реализации, чем предыдущее упражнение с функцией `ParseIni()`.



5. Процедурное программирование

Цель этой главы – завершить тему процедурного программирования на языке Go, начатую в предыдущих главах. На языке Go можно писать программы исключительно в процедурном стиле, в объектно-ориентированном стиле или использовать комбинацию двух парадигм. Знание процедурного программирования совершенно необходимо, потому что оно лежит в основе объектно-ориентированного и параллельного программирования на языке Go.

В предыдущих главах были представлены встроенные типы данных в языке Go, на протяжении которых использовалось множество различных инструкций и управляющих структур языка Go, а также было создано много небольших функций. В этой главе более подробно будут рассматриваться инструкции и управляющие структуры языка Go, а также намного детальнее будут исследоваться вопросы создания и использования собственных функций. В табл. 5.1 приводится список встроенных функций языка Go, большинство из которых уже были описаны выше¹.

Некоторые сведения, о которых рассказывается в этой главе, менее формально уже были представлены в предыдущих главах, а некоторые касаются аспектов программирования на языке Go, которые будут рассматриваться в следующих главах. Везде, где это необходимо, будут даваться ссылки на соответствующие разделы.

5.1. Введение в инструкции

Формально во многих ситуациях синтаксис языка Go требует использовать *точки с запятой* (;) для завершения инструкций. Однако, как было показано выше, в действительности использовать точки с запятыми требуется довольно редко. Это объясняется тем,

¹ В табл. 5.1 отсутствуют встроенные функции `print()` и `println()`, так как их использование в программах нежелательно. Они существуют исключительно ради удобства разработчиков компилятора Go и могут быть удалены из языка. Вместо них следует использовать такие функции, как `fmt.Print()` (§3.5).

что компилятор автоматически вставляет точки с запятой в концах непустых строк, завершающихся идентификатором, числовым литералом, символьным литералом, строковым литералом, некоторыми ключевыми словами (`break`, `continue`, `fallthrough`, `return`), инструкциями инкремента или декремента (`++` или `--`) или закрывающей круглой, квадратной или фигурной скобкой (`()`, `[]`, `{}`).

Два типичных случая обязательного использования точек с запятой – когда в одной строке находятся две или более инструкций и в простых циклах `for` (§5.3).

Важным следствием автоматического добавления точек с запятой является то обстоятельство, что открывающие фигурные скобки не могут находиться в отдельной строке.

// Правильно	// НЕПРАВИЛЬНО! (Не будет компилироваться)
for i := 0; i < 5; i++ {	for i := 0; i < 5; i++
fmt.Println(i)	{
}	fmt.Println(i)
	}

Фрагмент справа не будет компилироваться, потому что компилятор автоматически добавит точку с запятой после `++`. Аналогично, если написать бесконечный цикл (`for`) с открывающей фигурной скобкой в следующей строке, компилятор вставит точку с запятой после инструкции `for`, и программный код снова не сможет быть скомпилирован.

Эстетика размещения фигурных скобок – постоянный источник споров, но только не в языке Go. Отчасти потому, что автоматическая расстановка точек с запятой сужает круг возможных местоположений фигурных скобок, а отчасти потому, что многие программисты на Go используют программу `gofmt`, форматирующую исходные тексты на языке Go стандартным способом. Фактически все исходные тексты в стандартной библиотеке отформатированы с помощью `gofmt`. Именно этим объясняется единообразие оформления программного кода, являющегося результатом работы разных программистов¹.

¹ На момент написания этих строк программа `gofmt` не поддерживала возможности переноса строк по достижении максимальной ширины и в некоторых случаях `gofmt` может объединять две или более строк с переносами, создавая одну длинную строку. Исходный программный код, демонстрируемый в книге, был автоматически извлечен из действующих примеров и программ и вставлен в готовый для тиражирования PDF-файл книги, где длина строки ограничивается 75 символами. Поэтому сначала программный код для книги обрабатывался программой `gofmt`, а затем длинные строки переносились вручную.

Таблица 5.1. Встроенные функции

Функция	Описание/результат
<code>append(s, ...)</code>	Срез <code>s</code> плюс новые элементы, добавленные в конец, если емкости среза <code>s</code> достаточно для размещения новых элементов; в противном случае создается новый срез, куда копируются срез <code>s</code> и новые элементы (см. §4.2.3, выше)
<code>cap(x)</code>	Емкость среза <code>x</code> , или емкость буфера канала <code>x</code> , или длина массива <code>x</code> ; см. также описание функции <code>len()</code> (§4.2 выше)
<code>close(ch)</code>	Закрывает канал <code>ch</code> (за исключением каналов, открытых только на прием). После закрытия в канал невозможно будет отправить данные. Данные по-прежнему могут приниматься из канала (например, отправленные ранее, но пока не принятые данные), а когда в канале не останется данных, при попытке чтения из канала будут возвращаться нулевые значения
<code>complex(r, i)</code>	Значение типа <code>complex128</code> с указанной действительной (<code>r</code>) и мнимой (<code>i</code>) частями, обе типа <code>float64</code> (см. §2.3.2.1 выше)
<code>copy(dst, src)</code> <code>copy(b, s)</code>	Скопирует элементы (возможно, перекрывающиеся) из среза <code>src</code> в срез <code>dst</code> , усекая копируемый фрагмент, если в целевом срезе недостаточно места; или скопирует байты из строки <code>s</code> в срез <code>b</code> типа <code>[]byte</code> (см. §4.2.3 выше и §6.3)
<code>delete(m, k)</code>	Удалит элемент с ключом <code>k</code> из отображения <code>m</code> или, если такой элемент отсутствует, ничего не сделает (см. §4.3 выше)
<code>imag(cx)</code>	Мнимая часть числа <code>cx</code> типа <code>complex128</code> как число типа <code>float64</code> (см. §2.3.2.1 выше)
<code>len(x)</code>	Длина среза <code>x</code> , или количество элементов в буфере канала <code>x</code> , или длина массива <code>x</code> , или число элементов в отображении <code>x</code> , или число байт в строке <code>x</code> ; см. также описание функции <code>cap()</code> (§4.2.3 выше)
<code>make(T)</code> <code>make(T, n)</code> <code>make(T, n, m)</code>	Ссылка на срез, отображение или канал типа <code>T</code> . Если указано значение <code>n</code> , оно определяет длину и емкость среза, или первоначальную емкость отображения, или размер буфера канала. Только для срезов могут указываться значения <code>n</code> и <code>m</code> , определяющие длину и емкость (см. §4.2, где описываются срезы, §4.3, где описываются отображения, и главу 7, где описываются каналы)
<code>new(T)</code>	Указатель на значение типа <code>T</code> (см. §4.1 выше)
<code>panic(x)</code>	Генерирует перехватываемое исключение со значением <code>x</code> (см. §5.5.1 ниже)
<code>real(cx)</code>	Действительная часть числа <code>cx</code> типа <code>complex128</code> как число типа <code>float64</code> (см. §2.3.2.1 выше)
<code>recover()</code>	Перехватывает исключение (см. §5.5.1 ниже)

В языке Go поддерживаются операторы инкремента (`++`) и декремента (`--`), перечисленные в табл. 2.4 (выше). Оба они имеют только постфиксную форму записи, то есть следуют за операндом и не имеют возвращаемого значения. Эти ограничения не позволяют использовать операторы в качестве выражений и избавляют от неоднозначных ситуаций. Например, эти операторы невозможно применить к аргументам функций или записать такое выражение: `i = i++` (допустимое в языках C и C++, где результат зависит от реализации компилятора).

Присваивание значений выполняется с помощью *оператора присваивания* `=`. Переменные могут одновременно объявляться с помощью ключевого слова `var` и инициализироваться с использованием оператора `=`, например: инструкция `var x = 5` создаст новую переменную `x` типа `int` и присвоит ей значение 5. (То же самое можно получить с помощью инструкций `var x int = 5` и `x := 5`.) Тип, присваиваемый переменной, должен быть совместим с присваиваемым значением. Если оператор `=` используется без ключевого слова `var`, переменная слева от него уже должна существовать к этому моменту. Допускается указывать слева от оператора присваивания несколько переменных, перечисленных через запятую, а также использовать *пустой идентификатор* (`_`), совместимый со значениями любых типов и позволяющий игнорировать любые присваиваемые значения. Множественное присваивание упрощает возможность *обмена двух переменных значениями* без явного создания временной переменной, например `a, b = b, a`.

Оператор *сокращенного объявления переменных* (`:=`) одновременно объявляет новую переменную и присваивает ей значение. Допускается указывать слева от оператора несколько переменных, перечисленных через запятую, как в случае с оператором присваивания `=`, за исключением того, что слева должна быть указана как минимум одна новая переменная (пустой идентификатор не считается). Если слева от оператора указана уже существующая переменная, ей просто будет присвоено новое значение, если только оператор `:=` не находится в начале новой области видимости, такой как инструкция `if` или блок инициализации инструкции `for` (§5.2.1 и §5.3).

`a, b, c := 2, 3, 5`

```
for a := 7; a < 8; a++ { // переменная a цикла затенит внешнюю переменную, a
    b := 11           // переменная b затенит внешнюю переменную b
    c = 13            // c - это внешняя переменная c
```

```

    fmt.Printf("inner: a→%d b→%d c→%d\n", a, b, c)
}
fmt.Printf("outer: a@%d b@%d c@%d\n", a, b, c)
inner: a→7 b→11 c→13
outer: a→2 b→3 c→13

```

Этот фрагмент демонстрирует способность оператора `:=` «затенять» переменные. Здесь, внутри цикла `for`, переменные `a` и `b` затеняют переменные из внешней области видимости, и, хотя это вполне допустимо, такой прием практически всегда является ошибкой программирования. С другой стороны, здесь существует только одна переменная `c` (из внешней области видимости), то есть такое ее использование очевидно и правильно. Использование переменных, затеняющих другие переменные, может быть весьма удобно, как будет показано чуть ниже, но небрежное использование этого приема может порождать проблемы.

Как будет обсуждаться далее в этой главе, инструкция `return` позволяет возвращать из функций одно или более *именованных* возвращаемых значений, без явного их указания в инструкции. В таких ситуациях возвращаться будут именованные возвращаемые значения, инициализированные *нулевыми значениями* в начале функции, которые могут изменяться в теле функции с помощью оператора присваивания.

```

func shadow() (err error) { // ЭТА ФУНКЦИЯ НЕ КОМПИЛИРУЕТСЯ!
    x, err := check1()      // x - создается; err - присваивается значение
    if err != nil {
        return // корректно вернет err
    }
    if y, err := check2(x); err != nil { // создаются y и внутренняя err
        return // внутренняя err затенит внешнюю err, поэтому по ошибке вернет nil!
    } else {
        fmt.Println(y)
    }
    return // вернет nil
}

```

Первая инструкция в функции `shadow()` создает переменную `x` и присваивает ей значение, но не создает переменную `err`, а просто присваивает ей новое значение, потому что она уже объявлена как возвращаемое значение функции `shadow()`. Это возможно потому, что оператор `:=` должен создать хотя бы одну непустую переменную,

и это условие соблюдается. То есть если переменная `err` не равна значению `nil`, она корректно будет возвращена вызывающей программой.

Простая инструкция в инструкции `if`, то есть необязательная инструкция, следующая за ключевым словом `if` и предшествующая условному выражению, начинает новую область видимости (§5.2.1). Поэтому здесь создаются *две* переменные, `y` и `err`, причем последняя *затеняет* внешнюю переменную `err`. Если переменная `err` не равна значению `nil`, возвращается значение внешней переменной `err` (то есть переменной `err`, объявленной как возвращаемое значение функции `shadow()`), которая имеет значение `nil`, присвоенное ей вызовом функции `check1()`, а значение, полученное в результате вызова `check2()`, было присвоено внутренней переменной `err`.

К счастью, проблема затенения переменных в этой функции во многом вымышленна, потому что компилятор Go прервет компиляцию и выведет сообщение об ошибке, если будет использована пустая инструкция `return`, когда одна из возвращаемых переменных окажется затенена другой, одноименной переменной. То есть данная функция просто не будет скомпилирована.

Простейшее решение проблемы заключается в том, чтобы добавить в начало функции строку с объявлениями переменных (например, `var x, y int` или `x, y := 0, 0`) и заменить оператор `:=` на оператор `=` при вызове функций `check1()` и `check2()`. (Такой подход использовался, например, в функции `americanise()`; §1.6.)

Другое решение состоит в том, чтобы использовать *неименованное возвращаемое значение*. Такой подход вынуждает явно указывать значение в инструкции `return`, то есть в данном примере первые две инструкции `return` превратятся в инструкции `return err` (они будут возвращать разные, но правильно выбранные переменные), и последняя превратится в инструкцию `return nil`.

5.1.1. Преобразование типа

Язык Go предоставляет возможность преобразования значений между различными совместимыми типами, и такие преобразования удобны и безопасны. При преобразованиях между нечисловыми типами потери точности отсутствуют. Но при преобразовании между числовыми типами может наблюдаться потеря точности или другие эффекты. Например, пусть имеется инструкция `x := uint16(65000)`, тогда при преобразовании `y := int16(x)` из-за того что значение `x`

находится за пределами диапазона представления типа `int16`, переменная `y` получит, вероятно неожиданное, значение `-536`.

Ниже приводится синтаксис преобразований:

*тип*Результата := Тип(выражение)

В случае с числами любое целое или вещественное число можно преобразовать в другое целое или вещественное число (возможно, с потерей точности, если целевой тип имеет более узкий диапазон представления, чем исходный). То же справедливо и при преобразовании комплексных чисел между типами `complex128` и `complex64`. Преобразования между числовыми типами обсуждались в §2.3.

Строка может быть преобразована в срез типа `[]byte` (составляющие ее байты в кодировке UTF-8) или в срез типа `[]rune` (составляющие ее кодовые пункты Юникода), а срезы обоих типов, `[]byte` и `[]rune`, могут быть преобразованы в строку. Единственный символ, представленный значением типа `rune` (то есть `int32`), может быть преобразован в односимвольную строку. Преобразования строк и символов рассматривались в главе 3 (см. §3.2 и §3.3, а также табл. 3.2, 3.8 и 3.9).

Рассмотрим короткий пример, начав с определения простого пользовательского типа.

```
type StringSlice []string
```

Этот тип имеет также собственный метод `StringSlice.String()` (здесь не показан), возвращающий строковое представление среза `[]string` в форме, которую можно использовать для создания значения типа `StringSlice` с использованием синтаксиса составных литералов.

```
fancy := StringSlice{"Lithium", "Sodium", "Potassium", "Rubidium"}
fmt.Println(fancy)
plain := []string(fancy)
fmt.Println(plain)
StringSlice{"Lithium", "Sodium", "Potassium", "Rubidium"}
[Lithium Sodium Potassium Rubidium]
```

Вывод значения переменной `fancy` `StringSlice` производится с помощью собственной функции `StringSlice.String()`. Но после преобразования ее в значение типа `[]string` она выводится как любой

другой срез типа []string. (Создание собственных типов с их собственными методами рассматривается в главе 6.)

Преобразование других типов возможно, если тип выражения и тип, лежащий в основе требуемого Типа, совпадают или если выражение является нетипизированной константой, которая может быть представлена значением типа Тип, или если Тип является интерфейсом, который реализуется выражением¹.

5.1.2. Приведение типов

Множеством методов типа является множество всех методов, которые могут вызываться относительно значения этого типа. Для типов, не имеющих методов, это множество является пустым множеством. Тип `interface{}` в языке Go используется для представления пустого интерфейса, то есть значения типа, множество методов которого включает пустое множество. Поскольку любой тип имеет множество методов, включающее пустое множество (независимо от фактического количества методов), тип `interface{}` можно использовать для представления значения *любого* типа. Кроме того, значение типа `interface{}` можно преобразовать в значение фактического типа с помощью инструкции `switch` (§5.2.2.2), приведения типа или механизма интроспекции, реализованного в виде пакета `reflect` (§9.4.9)².

Переменные типа `interface{}` (или типов пользовательских интерфейсов) могут потребоваться при обработке данных, получаемых из внешних источников, когда желательно создать набор обобщенных функций и в объектно-ориентированном программировании. Одним из способов доступа к лежащему в основе значению является использование операции *приведения типа*, как показано ниже:

```
значениеТипа, логическое_значение := выражение.(Тип) // Контролируемое приведение
значениеТипа := выражение.(Тип) // Неконтролируемое; вызывает panic() при ошибке
```

¹ Также возможны и другие, менее очевидные преобразования – все они описываются в спецификации языка Go (golang.org/doc/go_spec.html).

² Программисты на Python могут рассматривать интерфейс `interface{}` как экземпляр объекта `object`, а программисты на Java – как экземпляр класса `Object`, хотя, в отличие от класса `Object` в языке Java, интерфейс `interface{}` можно использовать для представления не только встроенных, но и пользовательских типов. Программисты на C и C++ могут рассматривать `interface{}` как тип `void*`, который хранит информацию о фактическом типе значения.

В случае успеха контролируемого приведения типа возвращается значение выражения в виде значения указанного Типа и значение `true`, сообщающее об успехе. Если контролируемое приведение завершается неудачей (то есть тип выражения несовместим с указанным Типом), возвращается нулевое значение указанного Типа и `false`. Неконтролируемое приведение типа либо возвращает значение выражения в виде значения указанного Типа, либо вызывает встроенную функцию `panic()`, что ведет к завершению программы, если аварийную ситуацию своевременно не обработать. (Приемы возбуждения аварийных ситуаций и их обработки описываются ниже, в §5.5.)

Ниже приводится короткий пример, иллюстрирующий использование этих приемов.

```
var i interface{} = 99
var s interface{} = []string{"left", "right"}
j := i.(int) // j получит значение типа int (или будет вызвана функция panic())
fmt.Printf("%T→%d\n", j, j)
if i, ok := i.(int); ok {
    fmt.Printf("%T→%d\n", i, i) // i - затеняющая переменная типа int
}
if s, ok := s.([]string); ok {
    fmt.Printf("%T→%q\n", s, s) // s - затеняющая переменная типа []string
}
int→99
int→99
[]string→["left" "right"]
```

При приведении типа для получения результата часто принято использовать переменные, имена которых совпадают с именами оригинальных переменных, то есть *затеняющие переменные*. И в целом операция приведения типа используется, только когда ожидается, что выражение будет иметь определенный тип. (Если выражение может возвращать значение одного из нескольких типов, следует использовать инструкцию `switch` выбора по типу, §5.2.2.2.)

Обратите внимание, что при выводе оригинальных переменных `i` и `s` (обе имеют тип `interface{}`) они определяются как значения типов `int` и `[]string`. Это объясняется тем, что когда функции вывода из пакета `fmt` сталкиваются со значениями типа `interface{}`, они способны определять фактические значения.

5.2. Ветвление

В языке Go имеются три *инструкции ветвления*: `if`, `switch` и `select`; последняя из них будет обсуждаться позднее (§5.4.1). Эффект ветвления можно также получить с помощью отображения, ключи которого служат для выбора «ветви», а значения хранят ссылки на соответствующие функции. Подобный способ ветвления будет рассматриваться ниже, в этой же главе (§5.6.5).

5.2.1. Инструкция *if*

Инструкция `if` в языке Go имеет следующий синтаксис:

```
if необязательнаяИнструкция1; логическоеВыражение1 {  
    блок1  
} else if необязательнаяИнструкция2; логическоеВыражение2 {  
    блок2  
} else {  
    блок3  
}  
}
```

Инструкция может иметь нуль или более разделов `else if` и нуль или один завершающий раздел `else`. Каждый блок содержит нуль или более инструкций.

Фигурные скобки являются обязательными, а точки с запятой требуются, только когда присутствует *необязательная инструкция*. Необязательная инструкция должна быть *простой инструкцией* в терминологии Go: это может быть единственное выражение, инструкция передачи данных в канал (с использованием оператора `<-`), инструкция инкремента или декремента, инструкция присваивания или инструкция сокращенного объявления переменной. Если в необязательной инструкции объявляются переменные (например, с помощью оператора `:=`), их область видимости простирается от точки объявления до конца всей инструкции `if`, то есть они существуют в блоках `if` или `else if`, если те существуют, а также во всех нижележащих разделах и прекращают свое существование в конце инструкции `if`.

Логическое выражение должно возвращать значение типа `bool`. В языке Go нелогические значения не преобразуются автоматически в логические, поэтому всегда следует использовать оператор сравнения, например `if i == 0`. (Логические операторы и операторы сравнения перечислены в табл. 2.3 выше.)

Ранее уже приводилось множество примеров использования инструкции `if`, и еще больше их приводится в оставшейся части книги. Тем не менее рассмотрим еще два маленьких примера. Первый демонстрирует применение необязательной инструкции, а второй иллюстрирует характерные особенности инструкции `if` в языке Go.

<pre>// Канонический вид if α := compute(); α < 0 { fmt.Printf("(%d)\n", - α) } else { fmt.Println(α) }</pre>	<pre>// Подробный! { α := compute() if α < 0 { fmt.Printf("(%d)\n", - α) } else { fmt.Println(α) } }</pre>
--	---

Эти два фрагмента выведут одно и то же. Чтобы ограничить область видимости переменной `α`, в правом фрагменте пришлось использовать дополнительную пару фигурных скобок, тогда как в левом фрагменте эта область ограничивается инструкцией `if` автоматически.

Второй пример инструкции `if` — функция `ArchiveFileList()`, взятая из примера `archive_file_list` (в файле `archive_file_list/archive_file_list.go`). Позже тело этой функции будет использоваться для сравнения инструкций `if` и `switch`.

```
func ArchiveFileList(file string) ([]string, error) {
    if suffix := Suffix(file); suffix == ".gz" {
        return GzipFileList(file)
    } else if suffix == ".tar" || suffix == ".tar.gz" || suffix == ".tgz" {
        return TarFileList(file)
    } else if suffix == ".zip" {
        return ZipFileList(file)
    }
    return nil, errors.New("unrecognized archive")
}
```

Программа в этом примере читает файлы, имена которых передаются в командной строке, и для тех из них, которые она способна обработать (`.gz`, `.tar`, `.tar.gz`, `.zip`), программа выводит имя файла и список файлов, содержащихся в архиве.

Обратите внимание, что область видимости переменной `suffix`, созданной в первом разделе инструкции `if`, распространяется на всю

инструкцию `if ...else if ...`, то есть она видима во всех ветках, так же как переменная α в предыдущем примере.

В эту функцию можно было бы добавить заключительный раздел `else`, но в языке Go обычно используется структура, представленная здесь: инструкция `if` и нуль или более разделов `else if`, каждый из которых завершается инструкцией `return`, и вся эта конструкция завершается простой инструкцией `return`, а не заключительным разделом `else` с инструкцией `return`.

```
func Suffix(file string) string {
    file = strings.ToLower(filepath.Base(file))
    if i := strings.LastIndex(file, "."); i > -1 {
        if file[i:] == ".bz2" || file[i:] == ".gz" || file[i:] == ".xz" {
            if j := strings.LastIndex(file[:i], ".");
                j > -1 && strings.HasPrefix(file[j:], ".tar") {
                return file[j:]
            }
        }
        return file[i:]
    }
    return file
}
```

Функция `Suffix()` представлена здесь для полноты картины: она принимает имя файла (которое может включать путь к нему) и возвращает *расширение имени файла*, где все символы преобразованы в нижний регистр, то есть последнюю часть имени, начинающуюся с точки. Если в имени файла нет символа точки, возвращается само имя файла (без пути к нему). Если имя файла оканчивается двойным расширением, таким как `.tar.bz2`, `.tar.gz` или `.tar.xz`, тогда возвращается такое двойное расширение.

5.2.2. Инструкция `switch`

Существуют две разновидности инструкции выбора `switch`: выбор по значению выражения и выбор по типу. Разновидность инструкции `switch` выбора по значению выражения хорошо знакома программистам на C, C++ и Java, тогда как инструкция выбора по типу является характерной только для языка Go. Обе разновидности синтаксически очень похожи, но, в отличие от языков C, C++ и Java, инструкция `switch` в Go не выполняет все ветки до первой инструкции `break` (то есть в Go не требуется завершать инструкцией

break каждый раздел case). Однако имеется возможность явно потребовать «проваливаться» в следующий раздел case с помощью инструкции fallthrough.

5.2.2.1. Выбор по значению выражения

Инструкция switch *выбора по значению выражения* в языке Go имеет следующий синтаксис:

```
switch необязательная Инструкция; необязательное Выражение {  
case список Выражений1: блок1  
...  
case список ВыраженийN: блокN  
default: блокD  
}
```

Точка с запятой обязательна при наличии необязательной инструкции, независимо от наличия необязательного выражения. Каждый блок состоит из нуля или более инструкций.

Если в инструкции switch отсутствует необязательное выражение, компилятор предполагает, что в качестве выражения используется значение true. Необязательная инструкция должна быть *простой инструкцией*, как в инструкции if (см. выше). Если в необязательной инструкции объявляются переменные (например, с помощью оператора :=), их область видимости простирается от точки объявления до конца всей инструкции switch, то есть они существуют во всех блоках case и в блоке default и прекращают свое существование в конце инструкции switch.

Наибольшая эффективность инструкции switch достигается, когда инструкции case упорядочены сверху вниз от наиболее вероятных вариантов к наименее вероятным. Хотя в действительности прирост производительности будет замечен только при большом количестве вариантов выбора и когда инструкция switch многократно выполняется в цикле. Поскольку выполнение автоматически *не* «проваливается» через границы разделов case, нет необходимости вставлять инструкцию break в конец каждого блока case. Если такое «проваливание» желательно, достаточно просто добавить инструкцию fallthrough. Раздел default является необязательным, но при его наличии он может находиться в любом месте внутри инструкции switch. Если не будет найдено совпадения ни с одним выражением в инструкциях case, будет выполнен блок default, если имеется. В

противном случае управление будет передано инструкции, следующей за инструкцией `switch`.

В каждой инструкции `case` должен иметься список из одного или более выражений, разделенных запятыми, типы значений которых совпадают с типом значения необязательного выражения в инструкции `switch`. Если необязательное выражение отсутствует, компилятор по умолчанию использует значение `true`, то есть значение типа `bool`, и в этом случае все выражения во всех инструкциях `case` должны возвращать значение типа `bool`.

Если в блоке `case` или `default` будет встречена инструкция `break`, инструкция `switch` немедленно прервет выполнение и передаст управление следующей за ней инструкции или, если в инструкции `break` указано имя метки, ближайшей вмещающей инструкции `for`, `switch` или `select`, где определена указанная метка.

Ниже приводится простой пример инструкции `switch`, не имеющей необязательной инструкции и необязательного выражения.

```
func BoundedInt(minimum, value, maximum int) int {  
    switch {  
        case value < minimum:  
            return minimum  
        case value > maximum:  
            return maximum  
    }  
    return value  
}
```

Поскольку здесь нет необязательного выражения, вместо него компилятор будет использовать значение `true`. То есть все выражения во всех инструкциях `case` должны возвращать значение типа `bool`. Здесь оба выражения используют логические операторы сравнения.

```
switch {  
case value < minimum:  
    return minimum  
case value > maximum:  
    return maximum  
default:  
    return value  
}  
panic("unreachable")
```

Выше приводится альтернативная реализация тела функции `BoundedInt()`. Инструкция `switch` теперь охватывает все возможные случаи, поэтому поток выполнения никогда не достигнет конца функции. Тем не менее компилятор Go требует, чтобы функция завершалась инструкцией `return` или вызовом функции `panic()`, поэтому здесь использована последняя, чтобы лучше выразить семантику функции.

Функция `ArchiveFileList()`, представленная в предыдущем разделе, использует инструкцию `if`, чтобы определить, какую функцию вызвать. Ниже приводится аналогичная версия на основе инструкции `switch`.

```
switch suffix := Suffix(file); suffix { // Простейшая и неканоническая реализация!
case ".gz":
    return GzipFileList(file)
case ".tar":
    fallthrough
case ".tar.gz":
    fallthrough
case ".tgz":
    return TarFileList(file)
case ".zip":
    return ZipFileList(file)
}
```

Эта инструкция `switch` имеет и необязательную инструкцию, и необязательное выражение. В данном случае значение выражения имеет тип `string`, поэтому все выражения во всех инструкциях `case` должны содержать одну или более строк, разделенных запятыми, для сопоставления. Здесь также использована инструкция `fallthrough`, чтобы обеспечить обработку всех `tar`-файлов вызовом одной и той же функции.

Область видимости переменной `suffix` простирается от начала инструкции `switch` через все разделы `case` (и распространялась бы на раздел `default`, если бы он имелся) и завершается в конце инструкции `switch`, где переменная `suffix` прекращает свое существование.

```
switch Suffix(file) { // Каноническая форма
case ".gz":
    return GzipFileList(file)
```

```
case ".tar", ".tar.gz", ".tgz":  
    return TarFileList(file)  
case ".zip":  
    return ZipFileList(file)  
}
```

Здесь приводится более компактная и каноническая версия предыдущей инструкции `switch`. Вместо инструкции и выражения здесь используется только выражение: функция `Suffix()` (представленная выше) возвращает строку. А вместо инструкций `fallthrough` для обработки всех `tar`-файлов в одну инструкцию `case` был помещен список всех допустимых расширений, перечисленных через запятую.

Инструкции `switch` выбора по значению выражения в языке Go обладают большей гибкостью, чем аналогичные им инструкции в языках C, C++ и Java, и во многих случаях могут использоваться вместо инструкций `if`, обеспечивая более компактную форму записи.

5.2.2.2. Выбор по типу

Как отмечалось в разделе, описывающем возможность приведения типа (§5.1.2), при использовании переменных типа `interface{}` часто бывает необходимо получить доступ к фактическому значению. Если тип значения известен заранее, можно воспользоваться операцией приведения типа, но если значение может иметь один из нескольких типов, предпочтительнее использовать инструкцию `switch` выбора по типу.

Инструкция `switch` *выбора по типу* в языке Go имеет следующий синтаксис:

```
switch необязательнаяИнструкция; переключательТипа {  
case списокТипов1: блок1  
...  
case списокТиповN: блокN  
default: блокD  
}
```

Необязательная инструкция здесь подчиняется тем же правилам, что и в инструкциях `switch` выбора по значению выражения и `if`. И инструкции `case` действуют точно так же, как в инструкции `switch` выбора по значению выражения, за исключением того, что списки выражений должны содержать имена одного или более типов, перечисленных через запятую. Инструкции `default` и `fallthrough`

действуют точно так же, как в инструкции `switch` выбора по значению выражения, и, как обычно, каждый блок должен содержать нуль или более инструкций.

Переключатель типа – это выражение, значением которого является тип. Если значение выражения присваивается оператором `:=`, будет создана переменная со *значением*, равным значению, участвующему в выражении переключателя типа, но тип этой переменной будет зависеть от инструкции `case`: в инструкции `case`, содержащей единственный тип в списке, переменная будет иметь этот тип, а в инструкциях `case`, содержащих в списке два или более типов, тип переменной будет соответствовать типу выражения в переключателе типа.

Вообще говоря, разновидность приведения типа, поддерживаемого инструкцией `switch` выбора по типу, не приветствуется программистами, использующими объектно-ориентированный стиль и предпочитающими полагаться на *полиморфизм*. В языке Go полиморфизм поддерживается посредством механизма динамической (утиной) типизации (как будет показано в главе 6), тем не менее иногда возникают ситуации, когда явное приведение типа оказывается просто необходимым.

Ниже приводятся пример вызова простой функции, выполняющей определение типа, и ее вывод.

```
classifier(5, -17.9, "ZIP", nil, true, complex(1, 1))
param #0 is an int
param #1 is a float64
param #2 is a string
param #3 is nil
param #4 is a bool
param #5's type is unknown
```

Функция `classifier()` использует простую инструкцию `switch` выбора по типу. Это функция с *переменным числом аргументов*, то есть способная принимать произвольное число аргументов. А поскольку тип аргумента указан как `interface{}`, ее аргументы могут иметь любые типы. (Функции, включая функции с переменным числом аргументов, рассматриваются далее в этой главе, в §5.6.)

```
func classifier(items ...interface{}) {
    for i, x := range items {
        switch x.(type) {
            case bool:
```

```
        fmt.Printf("param #%d is a bool\n", i)
    case float64:
        fmt.Printf("param #%d is a float64\n", i)
    case int, int8, int16, int32, int64:
        fmt.Printf("param #%d is an int\n", i)
    case uint, uint8, uint16, uint32, uint64:
        fmt.Printf("param #%d is an unsigned int\n", i)
    case nil:
        fmt.Printf("param #%d is nil\n", i)
    case string:
        fmt.Printf("param #%d is a string\n", i)
    default:
        fmt.Printf("param #%d's type is unknown\n", i)
    }
}
```

Используемый здесь переключатель типа имеет тот же формат, что и операция приведения типа, то есть переменная. (Тип), а ключевое слово `type`, используемое вместо фактического имени типа, означает *любой* тип.

Иногда бывает необходимо получить доступ к фактическому значению переменной типа `interface{}` и определить ее действительный тип. Сделать это можно с помощью операции присваивания (оператора `:=`) в переключателе типа, как будет показано чуть ниже.

Одним из типичных случаев применения приведения типа является обработка данных, получаемых из внешних источников. Например, при анализе данных, полученных в формате JSON (JavaScript Object Notation – форма записи объектов JavaScript), необходимо как-то преобразовать данные в соответствующие типы данных языка Go. Выполнить это можно с помощью функции `json.Unmarshal()`. Если передать этой функции указатель на структуру с полями, соответствующими данным в формате JSON, она заполнит поля структуры, преобразуя каждый элемент данных в формате JSON в тип данных, соответствующий полю структуры. Но если структура данных в формате JSON заранее неизвестна, мы не сможем передать функции `json.Unmarshal()` указатель на конкретную структуру. В таких ситуациях можно передать ей указатель на значение типа `interface{}`, где `json.Unmarshal()` сохранит ссылку на отображение `map[string]interface{}`, в котором роль ключей будут играть имена полей в исходных данных, а роль значений – соответствующие значения типа `interface{}`.

Ниже приводится пример анализа объекта в формате JSON с неизвестной структурой, а также создания и вывода соответствующего строкового представления этого объекта.

```
MA := []byte(`{"name": "Massachusetts", "area": 27336, "water": 25.7,
              "senators": ["John Kerry", "Scott Brown"]}`)

var object interface{}
if err := json.Unmarshal(MA, &object); err != nil {
    fmt.Println(err)
} else {
    jsonObject := object.(map[string]interface{}) ❶
    fmt.Println(jsonObjectAsString(jsonObject))
}
{"senators": ["John Kerry", "Scott Brown"], "name": "Massachusetts",
"water": 25.700000, "area": 27336.000000}
```

Если в процессе анализа не возникло никаких ошибок, переменная `object` типа `interface{}` будет ссылаться на значение типа `map[string]interface{}`, в котором роль ключей будут играть имена полей объекта JSON. Функция `jsonObjectAsString()` принимает отображение данного типа и возвращает соответствующую строку в формате JSON. Для преобразования значения переменной `object` типа `interface{}` в значение переменной `jsonObject` типа `map[string]interface{}` здесь используется неконтролируемое приведение типа (§5.1.2; ❶). (Обратите внимание, что вывод, представленный здесь, был разбит на две строки, чтобы уместить по ширине книжной страницы.)

```
func jsonObjectAsString(jsonObject map[string]interface{}) string {
```

```
var buffer bytes.Buffer
buffer.WriteString("")
comma := ""
for key, value := range jsonObject {
    buffer.WriteString(comma)
    switch value := value.(type) { // затеняющая переменная ❶
    case nil: ❷
        fmt.Fprintf(&buffer, "%q: null", key)
    case bool:
        fmt.Fprintf(&buffer, "%q: %t", key, value)
    case float64:
        fmt.Fprintf(&buffer, "%q: %f", key, value)
    case string:
        fmt.Fprintf(&buffer, "%q: %q", key, value)
    case []interface{}:
```

```
fmt.Fprintf(&buffer, "%q: [", key)
innerComma := ""
for _, s := range value {
    if s, ok := s.(string); ok { // затеняющая переменная ❶
        fmt.Fprintf(&buffer, "%s%q", innerComma, s)
        innerComma = ", "
    }
}
buffer.WriteString("]")
}
comma = "", "
}
buffer.WriteString("")
return buffer.String()
}
```

Эта функция преобразует отображение, представляющее объект JSON, и возвращает строку в формате JSON, содержащую данные из объекта. Массивы в формате JSON внутри отображения представлены значениями типа `[]interface{}`. Функция делает одно простое предположение, касающееся JSON-массивов: они могут содержать только строковые элементы.

Для доступа к данным используется цикл `for ...range` (§5.3), выполняющий итерации по ключам и значениям отображения и использующий инструкцию `switch` выбора по типу способа обработки данных различных типов. Переключатель типа в инструкции `switch` (❶) присваивает значение `value` (типа `interface{}`) *новой* переменной `value` с типом, зависящим от соответствующей инструкции `case`. Это одна из ситуаций, когда имеет смысл использовать затеняющую переменную (хотя ничто не мешает создать новую переменную с другим именем). То есть, если `interface{}` `value` будет иметь значение типа `bool`, внутренняя переменная `value` получит тип `bool` и будет соответствовать второй инструкции `case`, то же справедливо и для других случаев.

Для записи значений в буфер используется функция `fmt.Fprintf()`, более удобная, чем, например, `buffer.WriteString(fmt.Sprintf(...))` (❷). Функция `fmt.Fprintf()` выводит данные в значение, реализующее интерфейс `io.Writer`, переданное в первом аргументе. Значение типа `bytes.Buffer` *не поддерживает* интерфейса `io.Writer`, но его *поддерживает* значение типа `*bytes.Buffer`, именно по этой причине функции передается адрес переменной `buffer`. Подробнее об этом рассказывается в главе 6, но если вкратце: `io.Writer` – это интерфейс,

в роли которого может выступать любое значение, имеющее подходящий метод `Write()`. Метод `bytes.Buffer.Write()` принимает указатель на приемник (то есть значение типа `*bytes.Buffer`, а не `bytes.Buffer`), поэтому только тип `*bytes.Buffer` соответствует требуемому интерфейсу, что означает необходимость передачи функции `fmt.Fprintf()` адреса переменной `buffer`, а не самого значения `buffer`.

Для обработки JSON-массивов, содержащихся в объекте JSON, используется внутренний цикл `for ...range`, выполняющий итерации по элементам значения типа `[]interface{}` и использующий контролируемое приведение типа (❸), то есть элемент добавляется в вывод, только если он действительно является строкой. Здесь снова используется затеняющая переменная (на этот раз переменная `s` типа `string`), поскольку требуется вывести не значение типа `interface{}`, а значение, на которое оно ссылается. (Операции приведения типа рассматриваются выше, в §5.1.2.)

Разумеется, если бы структура объекта в формате JSON была известна заранее, программный код можно было бы существенно упростить. Для извлечения данных достаточно было бы создать структуру для их хранения и реализовать метод вывода содержимого этой структуры в строковой форме. Ниже приводится фрагмент, реализующий такой случай.

```
var state State
if err := json.Unmarshal(MA, &state); err != nil {
    fmt.Println(err)
}
fmt.Println(state)
{"name": "Massachusetts", "area": 27336, "water": 25.700000,
 "senators": ["John Kerry", "Scott Brown"]}
```

Этот фрагмент очень похож на предыдущий. Однако здесь нет необходимости использовать функцию `jsonObjectAsString()`. Вместо этого достаточно определить тип `State` и соответствующий метод `State.String()`. (Здесь вывод также был разбит на две строки, только чтобы уместить его по ширине книжной страницы.)

```
type State struct {
    Name      string
    Senators []string
    Water     float64
    Area      int
}
```

Эта структура ничем не отличается от структур, встречавшихся ранее. Однако обратите внимание, что имена всех полей *должны* начинаться с заглавных букв, чтобы сделать их общедоступными, поскольку функция `json.Unmarshal()` способна заполнять только общедоступные поля. Кроме того, несмотря на то что пакет `encoding/json` не различает числовых типов (все числа в формате JSON он интерпретирует как значения типа `float64`), тем не менее функция `json.Unmarshal()` заполняет числовые поля, учитывая их типы.

```
func (state State) String() string {
    var senators []string
    for _, senator := range state.Senators {
        senators = append(senators, fmt.Sprintf("%q", senator))
    }
    return fmt.Sprintf(
        `{"name": %q, "area": %d, "water": %f, "senators": [%s]}`,
        state.Name, state.Area, state.Water, strings.Join(senators, ", "))
}
```

Этот метод возвращает значение типа `State` в виде строки в формате JSON.

В большинстве программ на языке Go не требуется выполнять приведение типов и производить выбор по типу. И даже когда они, казалось бы, необходимы, их использования часто можно избежать. Один из примеров их использования, когда передается значение, поддерживающее один интерфейс, и требуется проверить наличие поддержки другого интерфейса. (Эта тема обсуждается в главе 6, например в §6.5.2.) Другой пример, когда требуется преобразовать данные, поступающие из внешних источников, в типы, поддерживаемые в языке Go. Чтобы упростить сопровождение, практически всегда лучше будет изолировать такой программный код от остальной программы. Это позволит работать программе исключительно в терминах типов данных языка Go, а любые изменения, связанные с изменением формата или типов внешних данных, будут носить локальный характер.

5.3. Инструкция цикла for

В языке Go имеются две разновидности инструкции `for`, простая инструкция `for` и `for ... range`. Их синтаксис приводится ниже:

```

for { // Бесконечный цикл
    блок
}
for логическоеВыражение { // Цикл while
    блок
}
for необязательнаяПредИнструкция; логическоеВыражение; необязательнаяПостИнструкция { ❶
    блок
}
for индекс, символ := range Строка { // Итерации по символам в Строке ❷
    блок
}
for индекс := range Строка { // Итерации по символам в Строке ❸
    блок // символ, размер := utf8.DecodeRuneInString(Строка[индекс:])
}
for индекс, элемент := range массивИлиСрез { // Итерации по массиву или срезу ❹
    блок
}
for index := range массивИлиСрез { // Итерации по массиву или срезу ❺
    блок // item := массивИлиСрез[index]
}
for ключ, значение := range Отображение { // Итерации по элементам отображения ❻
    блок
}
for ключ := range Отображение { // Итерации по элементам отображения ❼
    блок // значение := Отображение[ключ]
}
for элемент := range Канал { // Итерации по элементам в канале
    блок
}

```

Фигурные скобки являются обязательными, а точки с запятой необходимы только в случае использования пред- и постинструкций (❶) — обе должны быть простыми инструкциями. Если в необязательной инструкции объявляются переменные или сохраняются значения, возвращаемые ключевым словом `range` (например, с помощью оператора `:=`), их область видимости простирается от точки объявления до конца всей инструкции `for`.

Логическое выражение в простой инструкции `for` (❶) должно возвращать значение типа `bool`, потому что в языке Go не выполняется автоматическое преобразование нелогических значений. (Логические операторы и операторы сравнения перечислены табл. 2.3 выше.)

Вторая разновидность инструкции `for ...range` итераций по строкам (❸) возвращает индексы байтов. Для строки `"XabYcZ"` из 7-битных ASCII-символов будут перечислены индексы 0, 1, 2, 3, 4, 5. Но для строки `"Xαβγδζ"` с символами в кодировке UTF-8 будут перечислены индексы 0, 1, 3, 5, 6, 8. Первый вариант инструкции `for ...range` для итераций по строкам (❷) практически всегда удобнее второго (❸).

Вторая разновидность инструкции `for ...range` итераций по массивам или срезам (❺) для непустых срезов или массивов возвращает индексы элементов от 0 до `len(срез) - 1`. Данная форма, как и первая разновидность инструкции `for ...range` итераций по массивам или срезам (❹), часто используется на практике. Наличие этих двух инструкций объясняет, почему в программах на языке Go так редко используется простая инструкция `for` (❶).

Инструкции `for ...range` итераций по элементам *ключ/значение* (❻) и по ключам отображений (❼) возвращают элементы или ключи в произвольном порядке. Если необходимо получить данные в определенном порядке, можно воспользоваться второй формой инструкции (❼), чтобы заполнить срез ключами и отсортировать этот срез как необходимо (пример реализации такого подхода представлен в предыдущей главе, в §4.3.4). Другое решение заключается в использовании упорядоченной структуры данных, например упорядоченного отображения. Пример такого отображения представлен в следующей главе (§6.5.3).

Если какая-либо из форм (❷–❼) будет применена к пустой строке, массиву, срезу или отображению, тело цикла `for` просто не будет выполнено ни разу, и управление сразу перейдет к следующей инструкции.

Выполнение цикла `for` можно прервать в любой момент с помощью инструкции `break`, передающей управление инструкции, следующей за инструкцией `for`, или, если в инструкции `break` указана метка, ближайшей вмещающей инструкции `for`, `switch` или `select`, где определена эта метка. Имеется также возможность с помощью инструкции `continue` передать управление в начало инструкции `for`, условному выражению или ключевому слову `range`, чтобы принудительно запустить следующую итерацию (или завершить цикл).

Выше приводилось множество примеров использования инструкции `for`, включая инструкцию `for ...range` (§3.3, §4.4.1 и §4.4.2), бесконечные циклы (§1.5 и §1.7) и простой цикл `for` (§3.5.4), потребность в котором редко возникает в программах на языке Go,

потому что другие разновидности часто оказываются гораздо удобнее. И конечно же в оставшейся части книги будет представлено еще больше примеров циклов `for`, включая некоторые примеры, следующие далее в этой главе. Поэтому здесь ограничимся лишь одним небольшим примером.

Предположим, что имеются двумерные срезы (например, типа `[[[] int]`) и необходимо отыскать в них определенное значение. Выполнить такой поиск можно двумя способами. Оба используют вторую форму инструкции `for ...range` итераций по массивам или срезам (❹).

<pre>found := false for row := range table { for column := range table[row] { if table[row][column] == x { found = true break } } if found { break } }</pre>	<pre>found := false FOUND: for row := range table { for column := range table[row] { if table[row][column] == x { found = true break FOUND } } }</pre>
--	--

Метка — это идентификатор, за которым следует двоеточие. Оба фрагмента с успехом решают поставленную задачу, но фрагмент справа короче и понятнее, потому что в нем при обнаружении искомого значения (x) выполняется прерывание внутреннего цикла с переходом в начало внешнего цикла за счет использования инструкции `break` с меткой. Преимущества использования приема прерывания цикла с переходом по метке возрастают многократно, когда он применяется в серии глубоко вложенных циклов (например, при выполнении итераций по трехмерным структурам данных).

Метки могут применяться к циклам `for`, инструкциям `switch` и `select`. Обе инструкции, `break` и `continue`, позволяют указывать метки и могут использоваться внутри циклов `for`. Инструкции `break`, с метками или без них, можно также использовать внутри инструкций `switch` и `select`.

Метки могут играть роль самостоятельных инструкций, и в этом случае их можно использовать в инструкциях безусловного перехода `goto` (например: `goto label`). Если инструкция `goto` перепрыгнет какую-либо инструкцию, создающую переменную, дальнейшее

поведение программы будет сложно предсказать – если повезет, программа завершит работу аварийно, но может случиться так, что она продолжит работу и даст ложные результаты. Один из случаев применения инструкции `goto` – автоматическая генерация программного кода. В этой ситуации инструкция `goto` может оказаться весьма удобным инструментом, и необязательно приводить к проблеме превращения программного кода в спагетти. На момент написания этих строк инструкция `goto` встречалась более чем в 30 файлах с исходными текстами стандартной библиотеки языка Go, но ни в одном примере в этой книге инструкция `goto` не используется, и я настоятельно не рекомендую ее использовать¹.

5.4. Инструкции организации взаимодействий и параллельного выполнения

Механизмы организации взаимодействия и параллельного выполнения, имеющиеся в языке Go, подробно рассматриваются в главе 7, но для полноты охвата темы процедурного программирования здесь будет описан их базовый синтаксис.

Go-подпрограмма – это функция или метод, выполняющаяся независимо и параллельно с другими go-подпрограммами в программе. Любая программа на языке Go имеет как минимум одну go-подпрограмму – главную go-подпрограмму, в которой выполняется функция `main()` из пакета `main`. Go-подпрограммы больше похожи на легковесные потоки выполнения или сопрограммы, в том смысле что они могут создаваться в большом количестве (тогда как даже небольшое количество обычных потоков выполнения потребляет значительное количество системных ресурсов). Все go-подпрограммы выполняются в общем адресном пространстве, и в языке Go имеется ряд элементарных механизмов блокировки, обеспечивающих безопасное совместное использование данных go-подпрограммами. Однако при параллельном программировании на языке Go рекомендуется использовать подход, основанный на обмене данными, а не на их совместном использовании.

¹ Инструкция `goto` попала под всеобщее осуждение с момента появления в 1968 году известной статьи Эдсгера Дейкстры (Edsger Dijkstra) «Go-to statement considered harmful» («О вреде инструкции `goto`») (www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF).

Каналом в языке Go называется одно- или двунаправленный канал, используемый для организации обмена (то есть приема и передачи) данными между двумя или более го-подпрограммами.

Вместе го-подпрограммы и каналы обеспечивают легковесный (то есть масштабируемый) способ параллельного выполнения без использования разделяемой памяти и потому не требующий применения блокировок. Тем не менее, как и при применении других механизмов параллельного выполнения, создание параллельных программ требует особого внимания, а их сопровождение обычно сложнее, чем обычных, непараллельных программ. Большинство операционных систем прекрасно приспособлены для одновременного выполнения множества программ, и использование этой возможности способно уменьшить трудозатраты на сопровождение, когда, например, выполняются несколько программ (или несколько копий одной программы), каждая из которых оперирует своим набором данных. Хороший программист приступает к созданию параллельной программы, только когда такой подход имеет очевидные преимущества, перевешивающие бремя дальнейшего ее сопровождения.

Го-подпрограмма создается с помощью инструкции `go`, имеющей следующий синтаксис:

```
go функция(аргументы)
```

```
go func(параметры) { блок }(аргументы)
```

Инструкция `go` либо вызывает уже имеющуюся функцию, либо анонимную функцию, созданную на месте вызова инструкции. Функция может иметь нуль или более параметров, как любая другая функция, и если она имеет параметры, ей должны передаваться соответствующие аргументы, как при вызове любой другой функции.

Выполнение вызываемой функции начинается немедленно, но в отдельной го-подпрограмме, при этом текущая го-подпрограмма (то есть го-подпрограмма, где была встречена инструкция `go`) также немедленно продолжит выполнение со следующей инструкции. Таким образом, после выполнения инструкции `go` в программе будут выполняться, по меньшей мере, две го-подпрограммы, оригинальная (главная го-подпрограмма) и вновь созданная.

Очень редко бывают ситуации, когда достаточно запустить группу го-подпрограмм и дождаться их завершения, без необходимости организовывать взаимодействие между ними. Чаще го-подпрограммы

должны действовать сообща, а добиться этого проще всего, обеспечив их возможностью взаимодействий. Ниже демонстрируются синтаксические конструкции, применяемые для отправки и приема данных:

```
канал <- значение // Блокирующая передача
<-канал           // Прием данных и их уничтожение
x := <-канал      // Прием и сохранение данных
x, ok := <-канал  // Как и выше, полюс проверка - открыт ли канал и имеются ли данные
```

Неблокирующую передачу можно выполнить с помощью инструкции `select` и до определенной степени – с помощью буферизованных каналов.

Каналы создаются с помощью встроенной функции `make()`, как показано ниже:

```
make(chan Тип)
make(chan Тип, емкость)
```

Если емкость буфера не указана, будет создан *синхронный канал*, то есть операция передачи данных будет блокироваться, пока принимающая сторона не будет готова принять их. Если емкость указана, будет создан *асинхронный канал*, обмен данными через который будет происходить без блокировки, при условии что в буфере достаточно места для отправляемых данных и при приеме в канале будут иметься данные.

По умолчанию создаются *двунаправленные каналы*, но их можно сделать *однонаправленными*, например чтобы лучше выразить семантику, которую сможет воплотить компилятор. В главе 7 будет показано, как создавать однонаправленные каналы, и с того момента везде, где это возможно, будут использоваться однонаправленные каналы.

Теперь поместим обсуждавшийся выше синтаксис в контекст небольшого примера¹. В примере реализуется функция `createCounter()`, возвращающая канал, через который в ответ на запрос отправляется целое число. Первым будет принято начальное значение, переданное функции `createCounter()`, а каждое последующее значение будет на единицу больше предыдущего. Ниже показано, как можно создать два независимых канала (каждый из которых

¹ Этот пример родился благодаря статье в блоге Эндрю Джерранда (Andrew Gerrand), nf.id.au/concurrency-patterns-a-source-of-unique-numbe. (Да, в конце адреса действительно отсутствует буква «г».)

действует в собственной go-подпрограмме) и результаты, производимые ими.

```

counterA := createCounter(2) // counterA имеет тип chan int
counterB := createCounter(102) // counterB имеет тип chan int
for i := 0; i < 5; i++ {
    a := <-counterA
    fmt.Printf("A→%d, B→%d) ", a, <-counterB)
}
fmt.Println()
(A→2, B→102) (A→3, B→103) (A→4, B→104) (A→5, B→105) (A→6, B→106)

```

Здесь демонстрируются два способа приема данных из канала, только чтобы показать, как это делается. Первый способ присваивает принятое значение переменной, а второй – передает принятое значение функции в виде аргумента.

В главной go-подпрограмме выполняются два вызова функции `createCounter()`, и в каждом из них создается новая go-подпрограмма, причем обе они изначально заблокированы. Как только в главной go-подпрограмме производится попытка получить данные из какого-либо канала, тут же выполняется передача, и программа получает очередное значение. Затем передающая go-подпрограмма снова блокируется, ожидая нового запроса от принимающей стороны. Эти два канала «бесконечны», в том смысле что всегда готовы отправить очередное значение. (Разумеется, при достижении верхнего предела представления значений типа `int` произойдет переход к значению, равному нижнему пределу.) Как только из каждого канала будет получено по пять значений, они снова заблокируются и будут пребывать в готовности к дальнейшему использованию.

Как прекратить выполнение go-подпрограмм, использующихся в примере выше, как только они станут не нужны? Для этого необходимо, чтобы они прервали выполнение своего бесконечного цикла, то есть чтобы они прекратили отправку данных и закрыли свои каналы. Один из способов будет представлен в следующем подразделе, и конечно же еще больше их будет представлено в главе 7, посвященной вопросам параллельного программирования и более подробно охватывающей эту тему.

```

func createCounter(start int) chan int {
    next := make(chan int)
    go func(i int) {

```

```

    for {
        next <- i
        i++
    }
}(start)
return next
}

```

Эта функция принимает начальное значение и создает канал для передачи и приема целых чисел. Затем она запускает *анонимную функцию* в новой go-подпрограмме, передавая ей начальное значение. Функция выполняет бесконечный цикл, который в каждой итерации просто отправляет целое число и затем увеличивает его на единицу. Поскольку канал создан с нулевой емкостью, операция передачи блокируется, пока не будет получен запрос от принимающей стороны. Блокируется только анонимная функция в go-подпрограмме, а остальные go-подпрограммы в программе продолжают выполняться, как ни в чем не бывало. Как только go-подпрограмма будет запущена (и конечно же тут же окажется заблокирована), немедленно будет выполнена следующая инструкция в функции, возвращающая канал вызывающей программе.

В некоторых ситуациях может потребоваться запустить множество go-подпрограмм, и каждую с собственным каналом передачи данных. Контролировать взаимодействия с ними можно с помощью инструкции `select`.

5.4.1. Инструкция *select*

Инструкция `select` имеет следующий синтаксис¹:

```

select {
case передачаИлиПрием1: блок1
...
case передачаИлиПриемN: блокN
default: блокD
}

```

В инструкции `select` проверяется возможность выполнения каждой инструкции передачи или приема, в порядке сверху вниз. Если какие-нибудь из них могут быть выполнены (то есть не будут

¹ Инструкция `select` в языке Go не имеет ничего общего с POSIX-функцией `select()`, используемой для наблюдения за дескрипторами файлов. Этой цели служит функция `Select()` из пакета `syscall`.

заблокированы), из них *произвольно* выбирается одна для продолжения работы. Если ни одна не может быть выполнена (то есть выполнение любой из них приведет к блокировке), возможны два сценария дальнейшего развития событий. Если в инструкции имеется раздел `default`, он выполняется, и выполнение продолжается с инструкции, следующей за инструкцией `select`. Но если раздел `default` отсутствует, инструкция `select` блокируется, пока хотя бы одно из взаимодействий не станет возможным.

Из такой логики работы инструкции `select` вытекает следующее. Инструкция `select` без раздела `default` *блокируется* и сможет завершиться, только когда станет возможным одно из предусмотренных взаимодействий (прием или передача). Инструкция `select` с разделом `default` *не блокируется* и выполняется немедленно, либо выполнив одно из взаимодействий, либо, если ни один из каналов не готов к приему или передаче, выполнив раздел `default`.

Чтобы закрепить новые знания, ниже приводятся два коротких примера. Первый из них в значительной мере надуманный, но он дает неплохое представление о том, как действует инструкция `select`. Второй пример демонстрирует более практичный способ использования.

```
channels := make([]chan bool, 6)
for i := range channels {
    channels[i] = make(chan bool)
}
go func() {
    for {
        channels[rand.Intn(6)] <- true
    }
}()
```

В этом фрагменте создаются шесть двунаправленных каналов для обмена логическими значениями. Затем создается `go`-подпрограмма, выполняющая бесконечный цикл, в каждой итерации которого случайно выбирается один из каналов, и в него отправляется значение `true`. Разумеется, `go`-подпрограмма немедленно блокируется, так как каналы небуферизованные и пока еще не предпринимались попытки чтения данных из них.

```
for i := 0; i < 36; i++ {
    var x int
    select {
        case <-channels[0]:
```

```

        x = 1
    case <-channels[1]:
        x = 2
    case <-channels[2]:
        x = 3
    case <-channels[3]:
        x = 4
    case <-channels[4]:
        x = 5
    case <-channels[5]:
        x = 6
    }
    fmt.Printf("%d ", x)
}
fmt.Println()
6 4 6 5 4 1 2 1 2 1 5 5 4 6 2 3 6 5 1 5 4 4 3 2 3 3 5 3 6 5 2 2 3 6 2

```

В этом фрагменте шесть каналов используются для имитации бросания игрового кубика (та или иная грань которого, строго говоря, выпадает псевдослучайно)¹. Инструкция `select` ожидает, пока по одному из каналов не будет что-нибудь отправлено (инструкция `select` блокируется, потому что в ней отсутствует раздел `default`), и как только один или более каналов будет готов к передаче, псевдослучайно будет выбран один из разделов `case`. Так как инструкция `select` находится внутри простого цикла `for`, она будет выполнена фиксированное число раз.

Теперь рассмотрим более практичный пример. Предположим, что требуется выполнить одни и те же дорогостоящие вычисления с двумя отдельными наборами исходных данных и произвести две последовательности результатов. Ниже приводится заготовка функции, реализующей такие вычисления.

```

func expensiveComputation(data Data, answer chan int, done chan bool) {
    // подготовка ...
    finished := false
    for !finished {
        // вычисления ...
        answer <- result
    }
    done <- true
}

```

¹ Реализации функций генерации псевдослучайных чисел можно найти в пакетах `math/rand` и `crypto/rand`.

Функции передаются некоторые данные `data` для обработки и два канала. Канал `answer` предназначен для передачи результатов наблюдающему программному коду, а канал `done` — для передачи извещения о завершении вычислений.

```
// подготовка ...
const allDone = 2
doneCount := 0
answer $\alpha$  := make(chan int)
answer $\beta$  := make(chan int)
defer func() {
    close(answer $\alpha$ )
    close(answer $\beta$ )
}()
done := make(chan bool)
defer func() { close(done) }()
go expensiveComputation(data1, answer $\alpha$ , done)
go expensiveComputation(data2, answer $\beta$ , done)
for doneCount != allDone {
    var which, result int
    select {
    case result = <-answer $\alpha$ :
        which = 'a'
    case result = <-answer $\beta$ :
        which = 'b'
    case <-done:
        doneCount++
    }
    if which != 0 {
        fmt.Printf("%c→%d ", which, result)
    }
}
fmt.Println()
a→3 b→3 a→0 b→9 a→0 b→2 a→9 b→3 a→6 b→1 a→0 b→8 a→8 b→5 a→0 b→0 a→3
```

Здесь выполняется подготовка каналов, запускаются дорогостоящие вычисления, осуществляется контроль над ходом выполнения, а в конце выполняются заключительные операции, и при этом не требуется использовать механизм блокировок.

Этот фрагмент начинается с создания двух каналов для приема результатов, `answer α` и `answer β` , и канала `done` для слежения за событием завершения вычислений. Затем создается анонимная функция, закрывающая каналы, и ее вызов передается инструкции

defer, чтобы каналы были закрыты, когда они станут не нужны, то есть при завершении вменяющей функции. Далее запускаются дорогостоящие вычисления (в отдельных go-подпрограммах), при этом в каждом вызове передаются отдельный набор данных для обработки, свой канал для возврата результатов и общий канал done.

Обеим процедурам дорогостоящих вычислений можно было бы передать один и тот же канал для результатов, но тогда было бы невозможно отличить результаты, произведенные при обработке одного набора данных, от результатов, произведенных при обработке другого набора (что, конечно, может быть неважно). Если бы потребовалось использовать общий канал для передачи результатов и обеспечить идентификацию происхождения результатов, можно было бы создать единственный канал, оперирующий структурами, такими как `type Answer struct{ id, answer int }`.

После запуска дорогостоящих вычислений в отдельных go-подпрограммах (но тут же заблокированных из-за использования небуферизованных каналов) можно приступить к приему результатов. Каждая итерация цикла `for` начинается с очистки значений переменных `which` и `result`, и затем блокирующая инструкция `select` выполняет произвольный раздел `case`, готовый к продолжению операции. В случае готовности одного из каналов передачи результатов в переменную `which` записывается признак происхождения результатов, после чего признак и результат выводятся. В случае готовности канала `done` увеличивается на единицу счетчик `doneCount`. Когда этот счетчик достигнет числа запущенных процедур дорогостоящих вычислений, это будет означать, что все они завершились и можно прекратить выполнение цикла.

После выхода из цикла `for` можно быть уверенными, что обе go-подпрограммы, выполнявшие дорогостоящие вычисления, не передадут никаких данных (так как они завершили выполнение своих бесконечных циклов `for` по завершении). Возвращая управление вызывающей программе, функция закроет каналы благодаря инструкции `defer` и освободит все использовавшиеся ресурсы. После этого сборщик мусора сможет удалить сами go-подпрограммы, так как они прекратили выполнение и каналы, использовавшиеся ими, были закрыты.

Механизмы взаимодействий и параллельного выполнения в языке Go весьма гибки и универсальны, и более подробно о них рассказывается в главе 7.

5.5. Инструкция `defer` и функции `panic()` и `recover()`

Инструкция `defer` используется, чтобы отложить вызов функции или метода (или анонимной функции, созданной в месте вызова) до момента, непосредственно предшествующему возврату управления вызывающей программе, но после того, как будут вычислены возвращаемые значения (если они имеются). Это дает возможность изменить *именованные* возвращаемые значения внутри отложенной функции (например, присвоив им другие значения с помощью оператора `=`). Если в функции или методе используются несколько инструкций `defer`, они будут выполнены в порядке, обратном порядку их следования.

Чаще всего инструкция `defer` применяется для закрытия файлов, по завершении работы с ними, для закрытия ненужных больше каналов или для обработки аварийных ситуаций.

```
var file *os.File
var err error
if file, err = os.Open(filename); err != nil {
    log.Println("failed to open the file: ", err)
    return
}
defer file.Close()
```

Это выдержка из функции `updateFrequencies()` в программе `word-frequency`, обсуждавшейся в предыдущей главе (§4.4.2). Она демонстрирует типичный способ открытия файла и откладывания операции его закрытия в случае успеха.

Такой способ создания некоторого значения и откладывания вызова некоторой функции закрытия, освобождающей ресурсы, занятые этим значением, перед тем как это значение будет утилизировано сборщиком мусора, является стандартным в языке Go¹. Разумеется, этот прием можно использовать и при работе со значениями пользовательских типов, реализующих функцию `Close()` или `Cleanup()`,

¹ В C++ для освобождения ресурсов используются деструкторы. В Java и Python процедура освобождения ресурсов связана с некоторыми проблемами, так как в этих языках не гарантируется вызов соответствующих методов `finalizer()/__del__()` объектов.

которую можно было бы передать инструкции defer, хотя на практике необходимость в этом возникает достаточно редко.

5.5.1. Функции panic() и recover()

В языке Go имеется механизм *обработки исключений*, реализованный в виде встроенных функций panic() и recover(). Эти функции можно использовать для реализации механизма исключений общего назначения, подобного имеющимся в некоторых других языках программирования (таких как C++, Java и Python), но такой подход считается неприемлемым в языке Go.

Ошибки в языке Go, когда что-то пошло неправильно (например, невозможно открыть файл) и есть возможность обработать эту ситуацию, отличаются от исключений, когда что-то в принципе «невозможно» (например, предварительное условие, которое всегда должно быть истинным, неожиданно стало ложным).

Характерный для языка Go способ обработки ошибок заключается в том, чтобы возвращать из функции или метода значение ошибки в последнем (или единственном) возвращаемом значении и всегда проверять все возвращаемые ошибки. (Практически единственный случай, когда общепринято игнорировать возвращаемые ошибки, – операция вывода в консоль.)

В случае «невозможной» ситуации можно вызвать встроенную функцию panic() с любым значением (например, строкой, описывающей проблему). В других языках для проверки подобных ситуаций можно использовать специальные инструкции¹, но в Go принято вызывать функцию panic(). На ранних стадиях разработки и до подготовки окончательной версии проще и, пожалуй, лучше вызывать функцию panic() для завершения программы, чтобы возникающие проблемы нельзя было игнорировать и вынудить программиста быстрее ликвидировать их. После развертывания приложения лучше избегать аварийного завершения приложения, насколько это возможно. Добиться этого можно, оставив на месте неликвидированные вызовы функции panic() и добавив отложенные вызовы функции recover(). С помощью функции recover() можно перехватывать и фиксировать в журнале любые аварийные ситуации (чтобы можно было обнаружить факт их появления) и возвращать вызывающему программному коду непустое значение ошибки, который затем

¹ Такие как инструкция assert. – *Прим. перев.*

сможет попытаться вернуть программу в нормальное состояние для продолжения работы.

Когда вызывается встроенная функция `panic()`, немедленно прекращается выполнение вызывающей функции или метода. Затем вызываются все отложенные функции или методы, как в случае нормального завершения функции. И наконец, управление передается вызывающей функции или методу, как если бы эта функция или метод вызвала функцию `panic()`, — теперь тот же процесс повторяется в вызывающей функции: ее выполнение прекращается, вызываются отложенные функции и т. д. Когда будет достигнута функция `main()`, ей некому возвращать управление, поэтому в этой точке программа завершается с выводом *трассировочной информации* в поток `os.Stderr`, включающей значение, переданное при вызове функции `panic()`.

Только что описанный процесс является обычным развитием событий в аварийных ситуациях. Однако если одна из отложенных функций или методов содержит вызов встроенной функции `recover()` (она может вызываться только в отложенных вызовах функций или методов), аварийное развитие событий прекратится. В этой точке можно выполнить любые операции по устранению аварии. Одно из решений состоит в том, чтобы просто игнорировать аварию. В этом случае управление будет передано вызывающей функции с отложенным вызовом функции `recover()`, которая продолжит работу как обычно. Такой подход не рекомендуется к использованию, но если он применяется, тогда следует хотя бы регистрировать информацию об аварии в журнале, чтобы факт появления проблемы не остался незамеченным. Другое решение заключается в выполнении операций по освобождению ресурсов и повторному вызову функции `panic()`, чтобы передать аварийную ситуацию дальше. Более универсальное решение предполагает создание значения ошибки и передачу его в виде последнего (или единственного) возвращаемого значения функции с отложенным вызовом `recover()`, то есть преобразование исключения (аварийной ситуации) в ошибку (в значение типа `error`).

Практически всегда в стандартной библиотеке вместо возбуждения аварийных ситуаций возвращаются значения ошибки. При создании собственных пакетов тоже лучше воздерживаться от использования функции `panic()` или хотя бы не позволять аварийной ситуации покидать пределы пакета, используя функцию `recover()`

для ее перехвата и возврата значения ошибки, как это делается в стандартной библиотеке.

Наглядным примером может служить пакет реализации регулярных выражений `regexp`. В своем составе он имеет несколько функций создания регулярных выражений, включая `regexp.Compile()` и `regexp.MustCompile()`. Первая из них возвращает скомпилированное регулярное выражение и значение `nil` или, если переданная строка не является допустимым регулярным выражением, значение `nil` и ошибку. Вторая возвращает скомпилированное регулярное выражение или возбуждает аварийную ситуацию. Первая функция идеально подходит для случаев, когда регулярные выражения поступают в программу из внешних источников (например, вводятся пользователем или извлекаются из файлов). Вторая лучше подходит, когда регулярное выражение включается непосредственно в текст программы, поскольку такой подход гарантирует, что в случае обнаружения ошибки в строке регулярного выражения, программа немедленно завершится.

В каких случаях следует позволить аварийной ситуации завершить программу, а в каких ликвидировать ее вызовом функции `recover()`? Здесь есть два противоречивых обстоятельства, которые следует учесть. Любой программист желал бы немедленного завершения его программы при возникновении логической ошибки, чтобы ее можно было устранить. Но точно так же любой программист желал бы, чтобы после развертывания его программы действовали безотказно.

Для поиска ошибок, которые могут быть обнаружены только во время выполнения программы (таких как недопустимые регулярные выражения), следует использовать функцию `panic()` (или функции, вызывающие ее, такие как `regexp.MustCompile()`), чтобы не допустить развертывания приложения, которое завершается аварийно сразу после запуска. Причем делать это следует только в функциях, которые *обязательно* будут вызываться в процессе выполнения. Например, в функции `init()` пакета `main` (если имеется), в функции `main()` пакета `main` и во всех функциях `init()` своих пакетов, импортируемых программой, плюс, разумеется, во всех функциях или методах, вызываемых этими функциями. Если используется комплект тестов, применение функции `panic()` можно распространить на все функции или методы, вызываемые тестами. Естественно, следует также убедиться, что такие потенциальные аварии всегда

возбуждаются, независимо от того, какими путями следует поток управления программы.

В функциях, которые при определенных обстоятельствах могут быть ни разу не вызваны в течение всего времени работы программы, следует использовать функцию `recover()`, если в них мы сами вызываем функцию `panic()`, или функции и методы, вызывающие ее, и превращать аварийную ситуацию в значение ошибки. В идеале функция `recover()` должна вызываться как можно ближе к соответствующему вызову функции `panic()`, где возможно ликвидировать аварию и восстановить нормальное состояние программы перед установкой значения ошибки, возвращаемого вмещающей функцией или методом. В функции `main()` в пакете `main` можно было бы организовать «ловушку» для любых аварийных ситуаций, в которой регистрировать все перехваченные аварии, но, к сожалению, в этом случае после выполнения отложенного вызова функции `recover()` программа все равно завершится. Впрочем, эту проблему можно обойти, как будет показано чуть ниже.

Далее будут представлены два примера. Первый демонстрирует, как превратить аварийную ситуацию в ошибку, а второй – как повысить устойчивость программы.

Представьте, что глубоко в недрах пакета имеется следующая функция, которую можно использовать, но нельзя изменить, потому что она входит в состав стороннего пакета, неподконтрольного нам.

```
func ConvertInt64ToInt(x int64) int {  
    if math.MinInt32 <= x && x <= math.MaxInt32 {  
        return int(x)  
    }  
    panic(fmt.Sprintf("%d is out of the int32 range", x))  
}
```

Эта функция преобразует значение типа `int64` в значение типа `int` или возбуждает аварийную ситуацию, если преобразование может дать неверный результат.

Прежде всего зачем в подобной функции может потребоваться возбуждать аварийную ситуацию? Чтобы, например, принудительно вызвать крах программы, как только что-то пойдет не так, и как можно раньше обнаружить ошибку программирования. Другая причина: когда имеется функция, вызывающая одну или более других функций и т.

д., где что-то может пойти не так, и необходимо, чтобы управление немедленно вернулось вызывающей функции. В подобном случае вызываемая функция возбуждает аварийную ситуацию, а вызывающая перехватывает ее (где бы она ни возникла) с помощью функции `recover()`. Обычно бывает желательно, чтобы функции из пакетов сообщали о проблемах, возвращая значение ошибки, а не возбуждая аварийную ситуацию. Поэтому внутри пакетов, где вызывается функция `panic()`, общепринято использовать функцию `recover()`, чтобы за границы пакета аварии попадали только в виде значений типа `error`. И еще одна причина: использование такого вызова, как `panic("unreachable")`, в местах, недостижимых для программы по логике размышлений (например, в конце функции, которая при любых обстоятельствах возвращает управление с помощью инструкций `return` до достижения конца), или вызов `panic()` при нарушении пред- или постусловия. Следование этим правилам гарантирует, что если когда-нибудь логика работы таких функций будет нарушена, программист сразу узнает об этом.

Если ни одна из вышеперечисленных причин не подходит, тогда следует избегать возбуждения аварийных ситуаций и при появлении проблем возвращать непустое значение ошибки. То есть в данном примере в случае успешного преобразования требуется вернуть значение типа `int` и `nil` и значения типа `int` и `error`, если преобразование потерпело неудачу. Ниже представлена функция-обертка, позволяющая достигнуть желаемого эффекта:

```
func IntFromInt64(x int64) (i int, err error) {  
    defer func() {  
        if e := recover(); e != nil {  
            err = fmt.Errorf("%v", e)  
        }  
    }()  
    i = ConvertInt64ToInt(x)  
    return i, nil  
}
```

В момент вызова этой функции возвращаемые значения будут инициализированы нулевыми значениями в соответствии с их типами, в данном случае 0 и `nil`. Если вызванная функция `ConvertInt64ToInt()` вернет управление как обычно, результат преобразования будет присвоен возвращаемому значению `i`, и вызывающей программе будет возвращено значение `i` и `nil` в качестве ошибки. Но если функция `ConvertInt64ToInt()` возбудит аварийную ситуацию, она будет

перехвачена отложенным вызовом анонимной функции, и возвращаемому значению `err` будет присвоена ошибка с текстом, извлеченным из текстового представления аварийной ситуации.

Как показывает функция `IntFromInt64()`, аварийная ситуация довольно просто преобразуется в ошибку.

Во втором примере демонстрируется, как повысить устойчивость веб-сервера к аварийным ситуациям. В главе 2 был представлен пример `statistics` (§2.4). Если бы при разработке этого сервера была допущена ошибка, например если бы мы случайно передали `nil` в `image.Image` и вызвали его метод, возникла бы аварийная ситуация, которая в отсутствие вызова функции `recover()` завершила бы работу программы. Такое поведение никуда не годится, если веб-сайт играет важную роль, особенно когда необходимо, чтобы он работал какое-то время без постоянного сопровождения. В данном случае требуется, чтобы сервер продолжал выполняться даже при появлении аварийных ситуаций и регистрировал любые аварии в журнале для последующего их анализа и устранения причин.

Измененная версия примера `statistics` (фактически решение `statistics_ans`) находится в файле `statistics_nonstop/statistics.go`. Как одно из изменений, для нужд тестирования на веб-страницу была добавлена кнопка **Panic!** (Авария!), щелчок на которой вызывает аварийную ситуацию. Но самым важным изменением является появившаяся у сервера способность пережить аварийную ситуацию. А чтобы помочь увидеть происходящее, была добавлена регистрация в журнале успешно обработанных запросов, ошибочных запросов и повторных запусков сервера. Ниже приводится короткий фрагмент типичного журнала.

```
[127.0.0.1:41373] served OK
[127.0.0.1:41373] served OK
[127.0.0.1:41373] bad request: '6y' is invalid
[127.0.0.1:41373] served OK
[127.0.0.1:41373] caught panic: user clicked panic button!
[127.0.0.1:41373] served OK
```

Пакету `log` было сообщено не добавлять текущее время сообщения в записи, чтобы сделать журнал более пригодным для демонстрации в книге.

Прежде чем перейти к обсуждению, вспомним, как выглядит оригинальный программный код.

```
func main() {
    http.HandleFunc("/", homePage)
    if err := http.ListenAndServe(":9001", nil); err != nil {
        log.Fatal("failed to start server", err)
    }
}

func homePage(writer http.ResponseWriter, request *http.Request) {
    // ...
}
```

Этот веб-сайт имеет единственную страницу, хотя представленный здесь прием легко можно распространить на многостраничные сайты. Если аварийная ситуация не будет перехвачена вызовом функции `recover()`, то есть если аварийная ситуация достигнет функции `main()`, сервер завершится. Другими словами, именно от этого и следует защитить сервер.

```
func homePage(writer http.ResponseWriter, request *http.Request) {
    defer func() { // Требуется добавить в каждую страницу
        if x := recover(); x != nil {
            log.Printf("[%v] caught panic: %v", request.RemoteAddr, x)
        }
    }()
    // ...
}
```

Чтобы обеспечить устойчивость веб-сервера к аварийным ситуациям, необходимо во все функции, выполняющие обработку страниц, добавить отложенный вызов анонимной функции, вызывающей функцию `recover()`. Это остановит дальнейшее распространение аварийной ситуации, но не уберет саму функцию обработки страницы от возврата (поскольку инструкция `defer` выполняется непосредственно перед завершением функции). Впрочем, это обстоятельство не имеет большого значения, потому что функция `http.ListenAndServe()` снова вызовет обработчик страницы, когда эта страница будет запрошена клиентом.

Конечно, для крупных веб-сайтов с огромным количеством обработчиков страниц добавление отложенного вызова функции, перехватывающей и регистрирующей аварийные ситуации, повлечет массовое дублирование программного кода, к тому же есть вероятность, что в какой-то обработчик отложенный вызов не будет добавлен по невнимательности. Эту проблему можно решить с помощью

функции-обертки, включающей программный код, необходимый для каждого обработчика. Благодаря такой *функции-обертке* можно будет убрать обработку аварийных ситуаций из обработчиков страниц, но при этом потребуется изменить вызовы функции `http.HandleFunc()`.

```
http.HandleFunc("/", logPanics(homePage))
```

Ниже приводится исходный текст оригинальной функции `homePage()` (то есть *не* имеющей отложенного вызова функции, вызывающей `recover()`), опирающейся на вызов функции-обертки `logPanics()` для обработки аварийных ситуаций.

```
func logPanics(function func(http.ResponseWriter,
    *http.Request)) func(http.ResponseWriter, *http.Request) {
    return func(writer http.ResponseWriter, request *http.Request) {
        defer func() {
            if x := recover(); x != nil {
                log.Printf("[%v] caught panic: %v", request.RemoteAddr, x)
            }
        }()
        function(writer, request)
    }
}
```

Эта функция принимает обработчик HTTP-запроса в виде единственного аргумента, создает и возвращает анонимную функцию, включающую отложенный вызов другой анонимной функции, перехватывающей и регистрирующей аварийные ситуации, и затем вызывает указанную функцию-обработчик. Такое решение дает тот же эффект, что и добавление отложенного вызова функции обработки и регистрации аварийных ситуаций, как было показано в модифицированной версии функции `homePage()`, но оно намного удобнее, потому что не требует добавлять отложенные вызовы функций во все обработчики страниц. Вместо этого достаточно просто передавать обработчики страниц функции `http.HandleFunc()` через вызов функции-обертки `logPanics()`.

Версию программы `statistics`, использующей это решение, можно найти в файле `statistics_nonstop2/statistics.go`. Подробнее об анонимных функциях рассказывается в следующем разделе, в подразделе, посвященном замыканиям (§5.6.3).

5.6. Пользовательские функции

Функции являются основой процедурного программирования, и язык Go обеспечивает самую широкую их поддержку. Методы в языке Go (рассматриваются в главе 6) очень похожи на функции, поэтому данный раздел в равной степени затрагивает процедурное и объектно-ориентированное программирование.

Ниже приводится фундаментальный синтаксис определения функции:

```
func имяФункции(необязательныеПараметры) типНеобязательногоВозвращаемогоЗначения {
    тело
}

func имяФункции(необязательныеПараметры) (необязательныеВозвращаемыеЗначения) {
    тело
}
```

Функция может принимать нуль или более параметров. Если функция не имеет параметров, должны указываться пустые круглые скобки. Если функция имеет один или более параметров, они должны записываться в форме списка: параметры1 тип1, ..., параметрыN типN, где параметры1 — имя единственного параметра или список из двух или более имен параметров указанного типа, разделенных запятыми. Параметры должны передаваться в указанном порядке: в языке Go отсутствуют аналоги именованных параметров в языке Python, хотя есть возможность добиться похожего эффекта, как будет показано ниже (§5.6.1.3).

Имени типа самого последнего параметра может предшествовать многоточие (...). Такие функции называют *функциями с переменным числом аргументов*, то есть функциями, способными принимать нуль или более значений указанного типа в виде значения параметра. Внутри функции такой параметр будет иметь тип []Тип.

Функция может возвращать нуль или более значений. Если функция ничего не возвращает, сразу вслед за круглой скобкой, закрывающей список параметров, должна следовать открывающая фигурная скобка. Если функция имеет одно неименованное возвращаемое значение, должен быть указан его тип. Если функция имеет два или более неименованных возвращаемых значения, их типы должны быть перечислены в круглых скобках, в форме списка: (тип1, ..., типN). Если функция имеет одно или более именованных возвращаемых значений, они должны быть перечислены в круглых скобках,

в форме списка: (значения1 тип1, ..., значенияN типN), где значения1 – единственное имя возвращаемого значения или список из двух или более имен возвращаемых значений, разделенных запятыми, указанного типа. Возвращаемые значения функции могут быть либо все именованные, либо все неименованные – смешивать их не допускается.

Функция, имеющая одно или более возвращаемых значений, должна содержать хотя бы одну инструкцию `return` или вызывать функцию `panic()` в последней инструкции. Если возвращаемые значения не имеют имен, инструкция `return` должна включать список всех возвращаемых значений соответствующих типов. Если возвращаемые значения имеют имена, инструкция `return` может включать список возвращаемых значений, подобно случаю с неименованными возвращаемыми значениями, либо вообще быть пустой (то есть без списка возвращаемых значений). Обратите внимание, что, несмотря на допустимость пустых инструкций `return`, использовать их не рекомендуется – ни один пример в книге не использует таких инструкций.

Если функция имеет одно или более возвращаемых значений, последней выполняемой инструкцией в ней должна быть инструкция `return` или вызов функции `panic()`. Компиляторы Go способны распознать, когда функция завершается аварийно и потому не сможет выполнить инструкцию `return`. К сожалению, имеющиеся на сегодняшний день компиляторы Go не понимают, что если функция завершается условной инструкцией `if` с безусловным разделом `else`, заканчивающимся инструкцией `return`, или инструкцией `switch` с разделом `default`, заканчивающимся инструкцией `return`, никаких последующих инструкций `return` не требуется. В таких случаях общепринято либо не использовать разделы `else` и `default` и помещать инструкцию `return` после инструкции `if` или `switch`, либо просто вставлять вызов `panic("unreachable")` в конец – оба этих способа были продемонстрированы в примерах выше (§5.2.2.1).

5.6.1. Аргументы функций

Выше уже приводилось множество примеров пользовательских функций, принимающих фиксированное число аргументов указанных типов. Используя параметр типа `interface{}`, можно создавать функции, принимающие аргументы любого типа. А используя параметр, типом которого является интерфейс (пользовательский или из

стандартной библиотеки), можно создавать функции, принимающие аргументы любых типов, обладающих определенными методами: подробнее об этой особенности рассказывается в главе 6 (§6.3).

В этом подразделе рассматривается еще одна возможность, касающаяся аргументов функции. В первом подподразделе демонстрируется, как использовать значения, возвращаемые функциями, непосредственно в качестве аргументов других функций. Во втором подподразделе демонстрируется, как создавать функции, принимающие переменное число аргументов. И в последнем подподразделе будут обсуждаться приемы создания функций, принимающих необязательные аргументы.

5.6.1.1. Вызовы функций в роли аргументов других функций

Если в программе имеется функция или метод с одним или более параметрами, ее можно вызвать с соответствующими аргументами. Но, кроме этого, вызываемой функции можно передать вызов другой функции или метода, при условии что количество значений, возвращаемых другой функцией или методом, точно соответствует количеству обязательных аргументов (и их типам).

Ниже приводится пример функции, которая принимает длины сторон треугольника (в виде трех значений типа `int`), вычисляет площадь треугольника по формуле Герона и возвращает ее.

```
for i := 1; i <= 4; i++ {
    a, b, c := PythagoreanTriple(i, i+1)
    Δ1 := Heron(a, b, c)
    Δ2 := Heron(PythagoreanTriple(i, i+1))
    fmt.Printf("Δ1 == %10f == Δ2 == %10f\n", Δ1, Δ2)
}
Δ1 == 6.000000 == Δ2 == 6.000000
Δ1 == 30.000000 == Δ2 == 30.000000
Δ1 == 84.000000 == Δ2 == 84.000000
Δ1 == 180.000000 == Δ2 == 180.000000
```

В этом фрагменте сначала вычисляются длины сторон прямоугольного треугольника по формуле Эвклида, а затем с помощью функции `Heron()`, принимающей три аргумента типа `int`, вычисляется его площадь по формуле Герона. После этого вычисления повторяются, только на этот раз функции `Heron()` передается

вызов функции `PythagoreanTriple()`, где на долю компилятора Go остается преобразование трех значений, возвращаемых функцией `PythagoreanTriple()`, в три аргумента функции `Heron()`.

```
func Heron(a, b, c int) float64 {
    α, β, γ := float64(a), float64(b), float64(c)
    s := (α + β + γ) / 2
    return math.Sqrt(s * (s - α) * (s - β) * (s - γ))
}

func PythagoreanTriple(m, n int) (a, b, c int) {
    if m < n {
        m, n = n, m
    }
    return (m * m) - (n * n), (2 * m * n), (m * m) + (n * n)
}
```

Функции `Heron()` и `PythagoreanTriple()` показаны здесь для полноты картины. Именованные возвращаемые значения в функции `PythagoreanTriple()` использованы исключительно для ее документирования.

5.6.1.2. Функции с переменным числом аргументов

Функцией с переменным числом аргументов называется такая функция, которая способна принимать нуль или более аргументов в своем последнем (или единственном) параметре. Такие функции отмечаются многоточием (...) непосредственно перед именем типа последнего или единственного параметра. Внутри функции подобный параметр становится срезом указанного типа. Например, допустим, что имеется функция с сигнатурой `Join(xs ...string) string`, тогда внутри функции параметр `xs` будет иметь тип `[]string`.

Ниже приводится короткий пример, демонстрирующий особенности использования функции с переменным числом аргументов, в данном случае это функция, возвращающая минимальное целое число из переданных ей. Для начала рассмотрим, как она вызывается и что возвращает.

```
fmt.Println(MinimumInt1(5, 3), MinimumInt1(7, 3, -2, 4, 0, -8, -5))
```

```
3 -8
```

Функции `MinimumInt1()` можно передать одно или более значений типа `int`, а она вернет наименьшее из них.

```
func MinimumInt1(first int, rest ...int) int {  
    for _, x := range rest {  
        if x < first {  
            first = x  
        }  
    }  
    return first  
}
```

Эту функцию легко можно было бы определить так, чтобы она могла вызываться вообще без аргументов, например `MinimumInt0(ints ...int)`, или по меньшей мере с двумя аргументами: `MinimumInt2(first, second int, rest ...int)`.

Если в программе уже имеется готовый срез со значениями типа `int`, с помощью функции `MinimumInt1()` можно отыскать наименьшее значение в этом срезе.

```
numbers := []int{7, 6, 2, -1, 7, -3, 9}  
fmt.Println(MinimumInt1(numbers[0], numbers[1:]...))  
-3
```

Функция `MinimumInt1()` требует передать единственный аргумент типа `int` и может еще принять нуль или более дополнительных аргументов. При вызове функции или метода с переменным числом аргументов можно после среза добавить оператор многоточия, благодаря чему срез превратится в последовательность из нуля или более аргументов, соответствующих элементам среза. (Этот прием обсуждался ранее, во время знакомства со встроенной функцией `append()`, в §4.2.3.) То есть здесь конструкция `numbers[1:]...` в вызове функции превратилась в последовательность отдельных параметров `6, -2, -1, 7, -3, 9`, которые внутри функции были сохранены в срезе `rest`. При использовании функции `MinimumInt0()`, упомянутой выше, вызов можно было бы упростить до `MinimumInt0(numbers...)`.

5.6.1.3. Функции с несколькими необязательными аргументами

В языке Go отсутствует непосредственная поддержка возможности создания функций с несколькими *необязательными аргументами*

различных типов. Однако этого эффекта легко добиться с помощью специализированной структуры и положиться на компилятор Go, всегда инициализирующий значения нулевыми значениями соответствующего типа.

Предположим, что имеется функция обработки некоторых пользовательских данных, которая по умолчанию обрабатывает все данные, но в иных ситуациях было бы желательно иметь возможность указывать первый и последний элементы для обработки, необходимость регистрации в журнале операций, выполняемых функцией, и передавать ей функцию обработки ошибок, возникающих при обработке недопустимых элементов.

Один из способов решения этой проблемы состоит в том, чтобы создать функцию с сигнатурой `ProcessItems(items Items, first last int, audit bool, errorHandler func(item Item))`. При такой организации значение 0 в параметре `last` будет означать, что индекс последнего элемента должен определяться автоматически, а функция `errorHandler` должна вызываться, только если она указана (то есть если этот параметр не равен `nil`). Это означает, что все вызовы, от которых требуется получить поведение по умолчанию, должны оформляться так: `ProcessItems(items, 0, 0, false, nil)`.

Более удачное решение заключается в том, чтобы определить функцию с сигнатурой `ProcessItems(items Items, options Options)`, где параметр типа `Options struct` будет хранить остальные параметры, по умолчанию имеющие нулевые значения. Это помогло бы сократить запись наиболее типичного вызова функции до: `ProcessItems(items, Options{})`. А в отдельных случаях, при необходимости указать один или более дополнительных параметров, это можно было бы сделать, определив значения отдельных полей структуры `Options`. (Полное описание структур можно найти в §6.4.) Рассмотрим, как все вышесказанное выглядит в программном коде, начав с определения структуры `Options`.

```
type Options struct {
    First    int // Первый элемент для обработки
    Last     int // Последний элемент для обработки
           // (0 означает обрабатывать все, начиная с элемента First)
    Audit    bool // Если true - регистрировать все операции
    ErrorHandler func(item Item) // Если не nil - вызывать для каждого
                       // недопустимого элемента
}
```

Структуры могут агрегировать или встраивать одно или более полей любых типов. (Разница между агрегированием и встраиванием описывается в главе 6.) Здесь структура `Options` агрегирует два поля типа `int`, поле типа `bool` и поле с функцией (ссылкой на функцию), имеющей сигнатуру `func(Item)`, где `Item` — некоторый пользовательский тип (в данном случае — тип одного элемента в значении пользовательского типа `Items`).

```
ProcessItems(items, Options{})
errorHandler := func(item Item) { log.Println("Invalid:", item) }
ProcessItems(items, Options{Audit: true, ErrorHandler: errorHandler})
```

В этом фрагменте демонстрируются два вызова пользовательской функции `ProcessItems()`. Первый вызов обрабатывает элементы с параметрами по умолчанию (то есть обрабатывает все элементы, не регистрирует операции и не вызывает обработчика ошибок для недопустимых элементов). Во втором вызове создается значение типа `Options`, в котором поля `First` и `Last` получают нулевые значения (тем самым функции предписывается обрабатывать все элементы), а поля `Audit` и `ErrorHandler` получают такие значения, что функция будет регистрировать в журнале все выполняемые ею операции и вызывать обработчик ошибок при обнаружении недопустимых элементов.

Данный прием использования структур для передачи необязательных аргументов применяется в стандартной библиотеке. Например, в функции `image.jpeg.Encode()`. Этот прием еще будет использоваться далее, в главе 6 (§6.5.2).

5.6.2. Функции *init()* и *main()*

В языке Go имеются два имени функций, зарезервированные для специальных целей: `init()` (во всех пакетах) и `main()` (только в пакете `main`). Эти две функции всегда должны быть определены как не принимающие аргументов и ничего не возвращающие. Пакет может иметь любое количество функций `init()`. Однако на момент написания этих строк существовал по меньшей мере один компилятор Go, поддерживавший лишь по одной функции `init()` на пакет, поэтому рекомендуется в каждом пакете использовать не более одной функции `init()`.

Функции `init()` в пакетах, как и функция `main()` в пакете `main`, вызываются автоматически, поэтому их не требуется вызывать явно. Для программ и пакетов функция `init()` является необязательной,

но всякая программа должна иметь единственную функцию `main()` в пакете `main`.

Инициализация и выполнение программ на языке Go всегда начинаются с пакета `main`. Если в этом пакете имеется раздел импортирования, каждый импортируемый пакет импортируется по очереди. Пакеты импортируются только один раз, даже если более чем один пакет импортирует один и тот же пакет. (Например, пакет `fmt` может импортироваться несколькими пакетами, но после того, как он будет импортирован в первый раз, повторное его импортирование производиться не будет, так как в этом нет необходимости.) Если пакет имеет собственный раздел импортирования, при его импортировании сначала будет произведен импорт указанных в нем пакетов. Затем будут созданы константы и переменные уровня пакета. А далее будет вызвана функция `init()` (если имеется). Когда в пакете `main` завершится импортирование пакетов (после того как они завершат импортирование своих пакетов и т. д.), в этот момент будут созданы константы и переменные пакета `main` и вызвана его функция `init()` (если имеется). И наконец, будет вызвана функция `main()` из пакета `main`, и программа начнет выполнение. Эта последовательность событий изображена на рис. 5.1.

В функциях `init()` допускается использовать инструкции `go`, но имейте в виду, что они запускаются до вызова функции `main.main()` и потому не должны зависеть от чего бы то ни было, что создается в функции `main()`.

Рассмотрим на примере (взятом из главы 1, файл `americanise/americanise.go`), как все происходит на практике.

```
package main
import (
    "bufio"
    "fmt"
    // ...
```

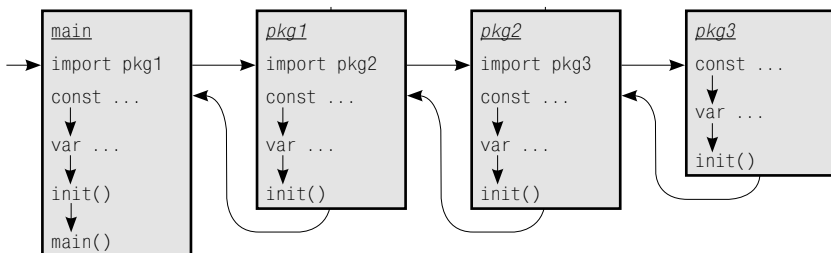


Рис. 5.1. Последовательность запуска программы

```
    "strings"
)
var britishAmerican = "british-american.txt"
func init() {
    dir, _ := filepath.Split(os.Args[0])
    britishAmerican = filepath.Join(dir, britishAmerican)
}
func main() {
    // ...
}
```

Выполнение программы на языке Go начинается с пакета `main`, а поскольку он имеет раздел импортирования, в первую очередь производится импортирование пакетов в указанном порядке, начиная с пакета `bufio`. Пакет `bufio` имеет собственный раздел импортирования, поэтому далее выполняется импорт этих пакетов. Всякий раз, когда в импортируемом пакете встречается его собственный раздел импортирования, сначала выполняется импортирование указанных в нем пакетов, затем создаются константы и переменные пакета и после этого вызываются его функции `init()`. После импортирования пакета `bufio` выполняется импортирование пакета `fmt`. Этот пакет импортирует `strings`, поэтому, когда дело доходит до импортирования пакета `strings` в пакете `main`, он пропускается, так как уже был импортирован.

По завершении импортирования создается переменная `britishAmerican`. Затем вызывается функция `init()` из пакета `main`. И наконец, вызывается функция `main()` из пакета `main` и начинается выполнение программы.

5.6.3. Замыкания

Замыкание — это функция, «захватывающая» константы и переменные, представленные в той же области видимости, где создается сама функция, если она ссылается на них. Это означает, что замыкание сохраняет возможность доступа к таким константам и переменным, даже когда оно вызывается далеко от того места, где оно было создано. При этом не имеет значения, находятся ли захваченные константы и переменные в области видимости в точке вызова замыкания, — если замыкание ссылается на них, они будут храниться в памяти для использования в замыкании.

В языке Go все анонимные функции (или *литералы функций*, как они называются в спецификации языка Go) являются замыканиями.

Замыкания создаются практически так же, как и обычные функции, но с одним важным отличием: замыкания не имеют имен (то есть за ключевым словом `func` сразу же следует открывающая круглая скобка). Чтобы иметь возможность пользоваться замыканиями, они обычно присваиваются переменным или сохраняются в структурах данных (таких как отображения или срезы).

Ранее в книге уже встречались примеры замыканий, например анонимные функции в инструкциях `defer` или `go` являются замыканиями. Замыкания также создавались и в других контекстах, например в функции `makeReplacerFunction()` в примере `americanise` (§1.6), при передаче анонимных функций в функции `strings.FieldsFunc()` и `strings.Map()` в главе 3 (§3.6.1), а также в функциях `createCounter()` (§5.4) и `logPanics()` (§5.5.1), представленных выше в этой главе. Тем не менее ниже будет рассмотрено несколько небольших примеров.

Одна из областей применения замыканий – создание *функций-обертков*, предопределяющих один или более аргументов обертываемой функции. Например, предположим, что к множеству различных имен файлов требуется добавить различные расширения. По сути, требуется обернуть оператор `+` конкатенации строк так, чтобы один операнд имел фиксированное значение (расширение), а другой (имя файла) мог изменяться.

```
addPng := func(name string) string { return name + ".png" }
addJpg := func(name string) string { return name + ".jpg" }
fmt.Println(addPng("filename"), addJpg("filename"))
filename.png filename.jpg
```

Здесь обе переменные, `addPng` и `addJpg`, хранят ссылки на анонимные функции (то есть на замыкания). Такие ссылки могут вызываться как обычные именованные функции, что демонстрирует фрагмент выше.

На практике, когда требуется создать множество похожих функций, вместо того чтобы создавать их по отдельности, часто используются *фабричные функции*, то есть функции, возвращающие другие функции. Ниже приводится пример фабричной функции, возвращающей функцию, которая добавляет указанное расширение к имени файла, но только если это расширение отсутствует в имени файла.

```
func MakeAddSuffix(suffix string) func(string) string {  
    return func(name string) string {  
        if !strings.HasSuffix(name, suffix) {  
            return name + suffix  
        }  
        return name  
    }  
}
```

Фабричная функция `MakeAddSuffix()` возвращает замыкание, сохраняющее значение переменной `suffix`, имевшееся на момент создания замыкания. Возвращаемое замыкание принимает один строковый аргумент (например, имя файла) и возвращает строку, которая является именем файла с указанным расширением.

```
addZip := MakeAddSuffix(".zip")  
addTgz := MakeAddSuffix(".tar.gz")  
fmt.Println(addTgz("filename"), addZip("filename"), addZip("gobook.zip"))  
  
filename.tar.gz filename.zip gobook.zip
```

В этом фрагменте демонстрируются создание двух замыканий, `addZip()` и `addTgz()`, и их использование.

5.6.4. Рекурсивные функции

Рекурсивная функция — это функция, вызывающая сама себя, а функции, вызывающие друг друга, называются *взаимно рекурсивными*. В языке Go имеется полная поддержка рекурсивных функций.

Рекурсивные функции обычно имеют одну и ту же структуру: они включают «точку выхода» и «тело». Точка выхода — обычно условная инструкция, такая как инструкция `if`, используемая для остановки рекурсии на основе одного из аргументов. Тело — где выполняются основные действия, включая по меньшей мере один вызов самой себя (или парной, взаимно рекурсивной функции), — в этот вызов должен передаваться аргумент со значением, отличным от текущего и который проверяется в условии выхода, чтобы обеспечить завершение рекурсии.

Рекурсивные функции упрощают обработку данных с рекурсивной структурой (таких как двоичные деревья), но они могут оказаться весьма неэффективными, например для применения в числовых вычислениях.

Начнем с очень простого (и неэффективного) примера, только чтобы понять, как выполняется рекурсия. Для начала посмотрим, как вызывается рекурсивная функция и что она возвращает, а затем изучим саму рекурсивную функцию.

```
for n := 0; n < 20; n++ {  
    fmt.Print(Fibonacci(n), " ")  
}  
fmt.Println()  
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

Функция `Fibonacci()` возвращает n -е число Фибоначчи.

```
func Fibonacci(n int) int {  
    if n < 2 {  
        return n  
    }  
    return Fibonacci(n-1) + Fibonacci(n-2)  
}
```

Инструкция `if` здесь играет роль условия выхода и гарантирует прекращение рекурсии. Это обеспечивается передачей в каждый рекурсивный вызов, выполняемый в теле функции (в инструкции `return`), уменьшенного значения аргумента n , благодаря чему в некоторый момент значение n обязательно окажется меньше двух.

Например, для вызова `Fibonacci(4)` условие выхода не будет выполнено, и функция вернет сумму двух рекурсивных вызовов, `Fibonacci(3)` и `Fibonacci(2)`. Первая из них, в свою очередь, вызовет `Fibonacci(2)` (которая вызовет `Fibonacci(1)` и `Fibonacci(0)`) и `Fibonacci(1)`, а вторая вызовет `Fibonacci(1)` и `Fibonacci(0)`. Для значений аргумента n меньше 2 возвращаются сами эти значения. Описанная последовательность вызовов показана на рис. 5.2.

Очевидно, что функция `Fibonacci()` выполняет слишком много повторяющихся вычислений даже для такого небольшого начального значения, как 4. Как избежать их, будет показано ниже (§5.6.7.1).

Мужские и женские последовательности Хофштадтера (Hofstadter Female and Male) – это последовательности целых чисел, вычисление которых основывается на взаимно рекурсивных функциях. Ниже приводится фрагмент, который выводит первые 20 значений в каждой последовательности:

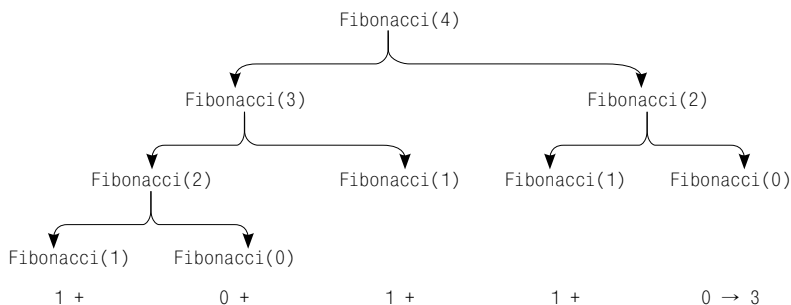


Рис. 5.2. Рекурсивное вычисление числа Фибоначчи

```

females := make([]int, 20)
males := make([]int, len(females))
for n := range females {
    females[n] = HofstadterFemale(n)
    males[n] = HofstadterMale(n)
}
fmt.Println("F", females)
fmt.Println("M", males)
F [1 1 2 2 3 3 4 5 5 6 6 7 8 8 9 9 10 11 11 12]
M [0 0 1 2 2 3 4 4 5 6 6 7 7 8 9 9 10 11 11 12]
  
```

И две взаимно рекурсивные функции, воспроизводящие эти последовательности.

```

func HofstadterFemale(n int) int {
    if n <= 0 {
        return 1
    }
    return n - HofstadterMale(HofstadterFemale(n-1))
}
func HofstadterMale(n int) int {
    if n <= 0 {
        return 0
    }
    return n - HofstadterFemale(HofstadterMale(n-1))
}
  
```

Каждая функция, как обычно, начинается с проверки условия выхода, чтобы обеспечить прекращение рекурсии, а в теле, где

происходит рекурсия, всегда передается уменьшенное значение, которое в конечном итоге удовлетворит условие выхода.

В некоторых языках программирования с функциями Хофштадтера могут возникать проблемы – они «спотыкаются» на том, что функция `HofstadterMale()` используется раньше, чем определяется. В таких языках требуется вставлять предварительное объявление функции `HofstadterMale()`. В языке Go отсутствуют такие ограничения – он позволяет определять функции в любом порядке.

Ниже приводится последний пример рекурсивной функции, которая выясняет, является ли указанное слово палиндромом (палиндромы – слова-перевертыши, читающиеся одинаково в обоих направлениях, такие как «PULLUP» и «ROTOR»).

```
func IsPalindrome(word string) bool {
    if utf8.RuneCountInString(word) <= 1 {
        return true
    }
    first, sizeOfFirst := utf8.DecodeRuneInString(word)
    last, sizeOfLast := utf8.DecodeLastRuneInString(word)
    if first != last {
        return false
    }
    return IsPalindrome(word[sizeOfFirst : len(word)-sizeOfLast])
}
```

Эта функция начинается с проверки условия выхода: если слово имеет нуль или один символ, значит, это палиндром, поэтому вызывающей программе возвращается `true` и рекурсия прекращается. В теле выполняется сравнение первого и последнего символов: если они отличаются, значит, слово не является палиндромом, поэтому можно немедленно прекратить рекурсию, вернув `false`. Но если первый и последний символы равны, тогда рекурсивно проверяется часть слова, заключенная между первым и последним символами (исключая их).

В случае со словом «PULLUP» функция сравнивает символы 'P' и 'P', затем рекурсивно вызывает саму себя, передавая строку "ULLU", и сравнивает символы 'U' и 'U', затем вызывает саму себя со строкой "LL", сравнивает символы 'L' и 'L', и, наконец, вызывает саму себя с пустой строкой. В случае со словом «ROTOR» функция сравнивает символы 'R' и 'R', затем рекурсивно вызывает саму себя, передавая строку "OTOT", и сравнивает символы 'O' и 'O', и, наконец, вызывает

саму себя со строкой "T". То есть в обоих случаях функция возвращает true. Но для слова «DECIDED» функция сравнит символы 'D' и 'D', затем вызовет рекурсивно саму себя со строкой "ECIDE" и сравнит символы 'E' и 'E', затем вызовет саму себя со строкой "CID", сравнит символы 'C' и 'D' и в этой точке вернет false.

В главе 3 (§3.6.3) говорилось, что функция `utf8.DecodeRuneInString()` возвращает первый символ (в виде значения типа `rune`) из указанной строки и количество байт, составляющих его. Функция `utf8.DecodeLastRuneInString()` действует аналогично, но возвращает последний символ. Два размера, полученные таким способом, можно безопасно использовать для извлечения подстроки, заключенной между крайними символами (то есть не опасаться, что в подстроку попадет часть многобайтного символа).

Когда в функции используется *хвостовая рекурсия*, то есть рекурсивный вызов является последней инструкцией, ее обычно можно преобразовать в простой цикл. Использование цикла позволяет сэкономить ресурсы, затрачиваемые на выполнение рекурсивных вызовов, хотя дополнительная проблема ограниченности памяти, выделяемой для стека, которая проявляется в некоторых языках при выполнении глубокой рекурсии, не так типична для программ на языке Go, благодаря его особенностям управления памятью. (В данном случае рекурсивную функцию `IsPalindrome()` можно преобразовать в цикл, что будет предложено сделать в упражнениях.) Конечно, в некоторых ситуациях рекурсия лучше всего подходит для реализации алгоритма, как будет показано в примере к главе 6, когда будет рассматриваться функция `omap.insert()`.

5.6.5. Выбор функции во время выполнения

Поскольку в языке Go функции являются обычными значениями, их (точнее, ссылки на них) можно сохранять в переменных, что делает возможным выбирать, какую функцию вызвать, во время выполнения. Кроме того, возможность создавать замыкания фактически позволяет создавать функции во время выполнения, то есть можно иметь две или более различных реализаций одной и той же функции (использующих разные алгоритмы), и создавать только одну из них для дальнейшего использования. В этом подразделе будут представлены оба подхода.

5.6.5.1. Ветвление с помощью отображений и ссылок на функции

В двух подразделах выше (§5.2.1 и §5.2.2.1) были представлены выдержки из разных версий пользовательской функции `ArchiveFileList()`, вызывающей другие функции, опираясь на расширение в имени файла. Первая версия функции использует инструкцию `if`, занимающую семь строк, а вторая – каноническую версию инструкции `switch`, занимающую пять строк. Но что будет происходить с ростом количества поддерживаемых расширений имен файлов? В версии с инструкцией `if` потребуется добавить две строки с инструкцией `else if` для каждого нового расширения. А в версии с инструкцией `switch` – одну строку для каждого нового расширения (или две, если форматировать исходные тексты с помощью `gofmt`). Если функция предназначена для использования в программе диспетчера файлов, в ней легко может потребоваться обрабатывать сотни расширений, что сделает функцию чересчур длинной.

```
var FunctionForSuffix = map[string]func(string) ([]string, error){
    ".gz": GzipFileList, ".tar": TarFileList, ".tar.gz": TarFileList,
    ".tgz": TarFileList, ".zip": ZipFileList}
func ArchiveFileListMap(file string) ([]string, error) {
    if function, ok := FunctionForSuffix[Suffix(file)]; ok {
        return function(file)
    }
    return nil, errors.New("unrecognized archive")
}
```

Эта версия функции использует отображение, ключами в котором являются строки (расширения имен файлов), а значениями – функции с сигнатурой `func(string) ([]string, error)`. (Все пользовательские функции: `GzipFileList()`, `TarFileList()` и `ZipFileList()` – имеют такую сигнатуру.)

Извлечение функции выполняется с помощью оператора индексирования `[]`, который отыскивает элемент, соответствующий указанному расширению, и присваивает переменной `ok` значение `true`; или возвращает `nil` и `false`, если расширение не является ключом отображения. Если требуемая функция найдена, она вызывается с указанным именем файла, и ее результат возвращается вызывающей программе.

Эта функция более масштабируема, чем использование инструкции `if` или `switch`, поскольку для нее не имеет значения, как много

элементов *расширение/функция* будет содержаться в отображении `FunctionForSuffix`, – сама функция от этого не изменится. И, в отличие от длинных инструкций `if` или `switch`, время поиска элемента в отображении с ростом их числа увеличивается не так быстро¹. Кроме того, применение отображения делает программный код более удобочитаемым и позволяет добавлять новые элементы динамически.

5.6.5.2. Динамическое создание функций

Другой сценарий, связанный с выбором функции во время выполнения, – когда имеется две или более функций с одинаковой функциональностью, но реализующих разные алгоритмы и было бы нежелательно производить выбор функции на этапе компиляции (например, чтобы дать пользователю возможность самому выбирать ту или иную реализацию с целью тестирования).

Например, для работы со строками, состоящими только из 7-битных ASCII-символов, можно написать более простую версию функции `IsPalindrome()`, представленной ранее (§5.6.4), и во время выполнения создавать ту версию функции, которая действительно необходима.

Реализовать это можно, объявив в пакете переменную с типом, совпадающим с сигнатурой функции, и затем создать соответствующую функцию в функции `init()`.

```
var IsPalindrome func(string) bool // Для хранения ссылки на функцию
func init() {
    if len(os.Args) > 1 &&
        (os.Args[1] == "-a" || os.Args[1] == "--ascii") {
        os.Args = append(os.Args[:1], os.Args[2:]...) // Отбросить аргумент
        IsPalindrome = func(s string) bool { // Простейшая ASCII-версия
            if len(s) <= 1 {
                return true
            }
            if s[0] != s[len(s)-1] {
                return false
            }
            return IsPalindrome(s[1 : len(s)-1])
        }
    }
}
```

¹ В ходе испытаний было установлено, что в высоконагруженной системе, на аппаратной платформе с 4-ядерным процессором AMD-64 и с тактовой частотой 3 ГГц, реализация на основе отображения действовала быстрее, чем реализация на основе инструкции `switch`, как только количество вариантов начинало превышать 50.

```
    }  
    } else {  
        IsPalindrome = func(s string) bool { // Версия для UTF-8  
            // ... то же, что и ранее ...  
        }  
    }  
}
```

Выбор той или иной реализации функции `IsPalindrome()` происходит на основе аргумента командной строки. Если аргумент указан, он удаляется из среза `os.Args` (благодаря чему остальная программа даже знать не будет о его существовании), и создается версия функции `IsPalindrome()` для работы со строками, состоящими из 7-битных ASCII-символов. Удаление аргумента из среза `os.Args` выглядит немного замысловато, потому что требуется сохранить в нем первый, третий и все последующие аргументы, кроме второго ("-a" или "--ascii"). Здесь нельзя использовать выражение `os.Args[0]` в вызове функции `append()`, потому что первым аргументом этой функции должен быть срез, поэтому здесь используется выражение `os.Args[:1]`, возвращающее срез с единственным элементом `os.Args[0]` (§4.2.1). Если флаг выбора ASCII-режима отсутствует, создается представленная ранее версия, которая корректно работает и с 7-битными ASCII-строками, и со строками Юникода в кодировке UTF-8. В остальной части программы функция `IsPalindrome()` может вызываться как обычно, хотя фактическая ее реализация будет отличаться в зависимости от выбранной версии. (Исходные тексты этого примера можно найти в файле `palindrome/palindrome.go`.)

5.6.6. Обобщенные функции

Выше в этой главе демонстрировалась функция поиска наименьшего целого числа среди переданных ей аргументов (§5.6.1.1). Алгоритм, использованный в этой функции, можно также применить к другим числовым типам и даже к строкам, поскольку его можно использовать для любых типов, поддерживающих оператор сравнения `<`. В C++ в таких случаях можно создать шаблонную функцию, параметризуемую типом результата, а компилятор сгенерирует все необходимые версии функции (то есть по одной для каждого используемого типа). В языке Go на момент написания этих строк не поддерживалась параметризация по типу результата, поэтому, чтобы добиться того же эффекта, требуется вручную создать все

необходимые функции (например, `MinimumInt()`, `MinimumFloat()`, `MinimumString()`). В результате в программе будет создано по одной функции для каждого типа (точно так же, как в C++, только в программе на языке Go каждая функция должна будет иметь уникальное имя).

Язык Go предлагает несколько альтернативных подходов, позволяющих избежать необходимости создавать функции, отличающиеся только типами данных, которыми они оперируют, хотя и за счет некоторой потери эффективности во время выполнения. Для небольших функций, используемых не слишком часто или которые выполняются достаточно быстро, альтернативные подходы могут оказаться весьма удобными.

Ниже приводятся несколько примеров использования обобщенной функции `Minimum()`.

```
i := Minimum(4, 3, 8, 2, 9).(int)
fmt.Printf("%T %v\n", i, i)
f := Minimum(9.4, -5.4, 3.8, 17.0, -3.1, 0.0).(float64)
fmt.Printf("%T %v\n", f, f)
s := Minimum("K", "X", "B", "C", "CC", "CA", "D", "M").(string)
fmt.Printf("%T %q\n", s, s)
int 2
float64 -5.4
string "B"
```

Функция возвращает значение типа `interface{}`, которое затем преобразуется во встроенный тип с помощью неконтролируемого приведения типа (§5.1.2).

```
func Minimum(first interface{}, rest ...interface{}) interface{} {
    minimum := first
    for _, x := range rest {
        switch x := x.(type) {
            case int:
                if x < minimum.(int) {
                    minimum = x
                }
            case float64:
                if x < minimum.(float64) {
                    minimum = x
                }
            case string:
```

```

    if x < minimum.(string) {
        minimum = x
    }
}
return minimum
}

```

Эта функция принимает как минимум одно значение (*first*) и нуль или более других значений (*rest*). Здесь используется тип `interface{}`, так как в языке Go этот тип можно использовать для представления любых других типов. Первоначально предполагается, что значение *first* является наименьшим, а затем выполняются итерации по значениям в срезе *rest*. При обнаружении значения меньше текущего минимального оно становится текущим минимальным. В конце функция возвращает минимальное значение типа `interface{}`, именно поэтому его необходимо преобразовывать во встроенный тип в точке вызова функции `Minimum()` с помощью неконтролируемого приведения типа.

Здесь все еще имеется повторяющийся код – блок инструкции `if` в каждом разделе `case`, но если бы повторяющегося кода было слишком много, можно было бы просто устанавливать логическую переменную в каждом разделе `case` (например, `change = true`) и за инструкцией `switch` добавить инструкцию `if change`, содержащую весь общий код.

Не вызывает сомнений, что функция `Minimum()` не так эффективна, как специализированные версии функции, оперирующие конкретными типами. Тем не менее об этой методике следует помнить, потому что она может пригодиться в ситуациях, когда неудобство и цена операции приведения типа – ничто, по сравнению с преимуществом возможности определить функцию всего один раз.

Проблема повторяющегося кода в обобщенной функции не имеет простого решения, когда один или более аргументов типа `interface{}` в действительности являются срезами. Например, ниже приводится функция, принимающая срез и элемент, тип которого совпадает с типом элементов в срезе, и возвращающая индекс первого вхождения элемента в срезе или `-1`, если искомый элемент отсутствует в срезе.

```

func Index(xs interface{}, x interface{}) int {
    switch slice := xs.(type) {
    case []int:

```

```

    for i, y := range slice {
        if y == x.(int) {
            return i
        }
    }
case []string:
    for i, y := range slice {
        if y == x.(string) {
            return i
        }
    }
}
return -1
}

```

Здесь реализована поддержка лишь двух типов, `int` и `string`, и в обоих случаях используется практически один и тот же программный код.

Ниже приводится пример использования функции `Index()`. (Программный код взят из тестовой программы `contains/contains.go`.)

```

xs := []int{2, 4, 6, 8}
fmt.Println("5 @", Index(xs, 5), " 6 @", Index(xs, 6))
ys := []string{"C", "B", "K", "A"}
fmt.Println("Z @", Index(ys, "Z"), "  A @", Index(ys, "A"))
5 @ -1 6 @ 2
Z @ -1 A @ 3

```

Все, что действительно необходимо, – это возможность обобщенной обработки срезов. Благодаря этому можно было бы оставить единственный цикл и все операции по приведению типов выполнять внутри него. Ниже демонстрируется функция, реализующая такую возможность и позволяющая получить тот же результат, что и предыдущая функция, достаточно лишь заменить во фрагменте выше вызов `Index()` на вызов `IndexReflectX()`.

```

func IndexReflectX(xs interface{}, x interface{}) int { // Более длинное решение
    if slice := reflect.ValueOf(xs); slice.Kind() == reflect.Slice {
        for i := 0; i < slice.Len(); i++ {
            switch y := slice.Index(i).Interface().(type) {
            case int:
                if y == x.(int) {
                    return i
                }
            }
        }
    }
    return -1
}

```

```

    }
    case string:
        if y == x.(string) {
            return i
        }
    }
}
return -1
}

```

Функция начинается с использования поддержки механизма рефлексии в языке Go (предоставляется пакетом `reflect`; §9.4.9) для преобразования значения `xs` типа `interface{}` в значение-срез `reflect.Value`. Это значение предоставляет методы, необходимые для итераций по элементам среза и извлечения произвольных элементов. Здесь доступ к каждому элементу осуществляется с помощью функции `reflect.Value.Interface()`, возвращающей значение типа `interface{}`, которое тут же присваивается переменной `y` внутри инструкции `switch` выбора по типу. Это гарантирует получение элемента фактического типа (`int` или `string`), который затем непосредственно сравнивается с результатом операции неконтролируемого приведения типа значения `x`.

В действительности пакет `reflect` способен на большее, благодаря чему эту функцию можно значительно упростить.

```

func IndexReflect(xs interface{}, x interface{}) int {
    if slice := reflect.ValueOf(xs); slice.Kind() == reflect.Slice {
        for i := 0; i < slice.Len(); i++ {
            if reflect.DeepEqual(x, slice.Index(i)) {
                return i
            }
        }
    }
    return -1
}

```

Данная версия опирается на использование функции `reflect.DeepEqual()`, выполняющей сравнение. Эта универсальная функция может также использоваться для сравнения массивов, срезов и структур.

Ниже приводится специализированная функция поиска индекса элемента в срезе.

```
func IntSliceIndex(xs []int, x int) int {  
    for i, y := range xs {  
        if x == y {  
            return i  
        }  
    }  
    return -1  
}
```

Эта версия выглядит намного проще, чем обобщенные версии, но такой подход требует создавать по одной функции для каждого используемого типа, где меняются лишь имена функций и названия типов в сигнатурах.

Имеется возможность с помощью пользовательских типов объединить преимущества обобщенного подхода к реализации единого алгоритма с простотой и эффективностью специализированных функций, о чем подробно рассказывается в следующей главе.

Ниже приводятся специализированная функция поиска индекса элемента в срезе типа `[]int` и обобщенная функция, которая используется специализированной версией для выполнения фактической работы.

```
func IntIndexSlicer(ints []int, x int) int {  
    return IndexSlicer(IntSlice(ints), x)  
}  
func IndexSlicer(slice Slicer, x interface{}) int {  
    for i := 0; i < slice.Len(); i++ {  
        if slice.EqualTo(i, x) {  
            return i  
        }  
    }  
    return -1  
}
```

Функция `IntIndexSlicer()` принимает срез типа `[]int`, где выполняется поиск, значение типа `int`, которое требуется найти, и передает их обобщенной функции `IndexSlicer()`. Функция `IndexSlicer()` оперирует значениями типа `Slicer`, где тип `Slicer` – пользовательский интерфейс, которому соответствуют любые значения, реализующие методы интерфейса `Slicer` (`Slicer.EqualTo()` и `Slicer.Len()`).

```
type Slicer interface {
    EqualTo(i int, x interface{}) bool
    Len() int
}

type IntSlice []int

func (slice IntSlice) EqualTo(i int, x interface{}) bool {
    return slice[i] == x.(int)
}

func (slice IntSlice) Len() int { return len(slice) }
```

Интерфейс Slicer определяет два метода, необходимых для реализации обобщенной функции IndexSlicer().

Тип IntSlice основан на типе []int (это объясняет, почему функция IntIndexSlicer() способна преобразовывать передаваемые ей значения типа []int в значения типа IntSlice без лишних сложностей) и реализует два метода, требуемые для поддержки интерфейса Slicer. Метод IntSlice.EqualTo() принимает индекс элемента в срезе и значение и возвращает true, если элемент с данным индексом равен указанному значению. Интерфейс Slicer определяет искомое значение как значение универсального типа interface{}, а не int, благодаря чему интерфейс может быть реализован любыми другими типами срезов (например, FloatSlice и StringSlice), поэтому значение требуется преобразовать в фактический тип. В данном случае можно смело использовать операцию неконтролируемого приведения типа, потому что значение поступает из функции IntIndexSlicer(), соответствующий аргумент типа int.

Можно реализовать другие пользовательские типы срезов, реализующие интерфейс Slicer, и использовать их с обобщенной функцией IndexSlicer().

```
type StringSlice []string

func (slice StringSlice) EqualTo(i int, x interface{}) bool {
    return slice[i] == x.(string)
}

func (slice StringSlice) Len() int { return len(slice) }
```

Единственное различие между StringSlice и IntSlice заключается в типах, на которых они основаны ([]string и []int соответственно), и в используемых операциях неконтролируемого приведения типа (string и int). Все сказанное выше относится и к типу FloatSlice (использующему типы []float64 и float64).

В последнем примере используется прием, продемонстрированный ранее, при обсуждении сортировки (§4.2.4), и он же применяется для реализации функций сортировки в пакете `sort` из стандартной библиотеки. Подробное обсуждение темы пользовательских интерфейсов и пользовательских типов приводится в главе 6.

При работе со срезами (или отображениями) часто бывает возможно создать обобщенные функции, в которых не требуется определять типы или выполнять приведение типов и для реализации которых не нужны пользовательские интерфейсы. Вместо этого можно превратить обобщенные функции в функции высшего порядка, абстрагируясь от всего, что связано с конкретными типами, как будет показано в следующем подразделе.

5.6.7. Функции высшего порядка

Функцией высшего порядка называется функция, принимающая в аргументах одну или более других функций и использующая их в своем теле.

Рассмотрим очень короткий пример простой функции высшего порядка, назначение которой трудно понять сразу.

```
func SliceIndex(limit int, predicate func(i int) bool) int {
    for i := 0; i < limit; i++ {
        if predicate(i) {
            return i
        }
    }
    return -1
}
```

Это – обобщенная функция, возвращающая индекс элемента в срезе, для которого функция `predicate()` вернет `true`. То есть эта функция в состоянии выполнить ту же работу, что и функции `Index()`, `IndexReflect()`, `IntSliceIndex()` и `IntIndexSlicer()`, обсуждавшиеся в предыдущем подразделе, причем без дублирования программного кода и без выбора по типу или операций приведения типов.

Функция `SliceIndex()` ничего не знает о типах срезов и элементов. В действительности она ничего не знает даже о срезах и элементах, которыми (косвенно) оперирует. Функция ожидает получить в первом аргументе длину среза, а во втором – функцию,

возвращающую логическое значение для той или иной позиции в срезе, указывающее на наличие искомого элемента в этой позиции.

Ниже приводятся пример четырех вызовов этой функции и полученные результаты.

```
xs := []int{2, 4, 6, 8}
ys := []string{"C", "B", "K", "A"}
fmt.Println(
    SliceIndex(len(xs), func(i int) bool { return xs[i] == 5 }),
    SliceIndex(len(xs), func(i int) bool { return xs[i] == 6 }),
    SliceIndex(len(ys), func(i int) bool { return ys[i] == "Z" }),
    SliceIndex(len(ys), func(i int) bool { return ys[i] == "A" }))
-1 2 -1 3
```

Анонимные функции, передаваемые функции `SliceIndex()` во втором аргументе, являются замыканиями, поэтому срезы, на которые они ссылаются (`xs` и `ys`), должны находиться в области видимости, где создаются эти функции. (Этот же прием используется в функции `sort.Search()` из пакета `sort`, входящего в состав стандартной библиотеки языка Go.)

В действительности `SliceIndex()` является универсальной функцией, которой вообще не требуется иметь дело со срезами.

```
i := SliceIndex(math.MaxInt32,
    func(i int) bool { return i > 0 && i%27 == 0 && i%51 == 0 })
fmt.Println(i)
459
```

В этом фрагменте функция `SliceIndex()` применяется для поиска наименьшего натурального числа, кратного числам 27 и 51. Вся хитрость здесь заключена в самом процессе поиска. Функция `SliceIndex()` выполняет итерации по числам от 0 до указанного предела (в данном случае `math.MaxInt32`) и в каждой итерации вызывает анонимную функцию (предикат). Как только предикат вернет `true`, функция `SliceIndex()` завершит итерации и вернет найденный индекс. В данном случае «индексом» является искомое натуральное число.

Помимо поиска по несортированным срезам, часто бывает желательно фильтровать их, отбрасывая элементы, не удовлетворяющие некоторому условию. Ниже приводится простой пример функции фильтрации высшего порядка в действии. Функция фильтрует срез

типа `[]int`, применяя переданную ей функцию, чтобы определить, какие элементы оставить, а какие отбросить.

```
readings := []int{4, -3, 2, -7, 8, 19, -11, 7, 18, -6}
even := IntFilter(readings, func(i int) bool { return i%2 == 0 })
fmt.Println(even)
[4 2 8 18 -6]
```

Здесь из среза `readings` удаляются нечетные числа.

```
func IntFilter(slice []int, predicate func(int) bool) []int {
    filtered := make([]int, 0, len(slice))
    for i := 0; i < len(slice); i++ {
        if predicate(slice[i]) {
            filtered = append(filtered, slice[i])
        }
    }
    return filtered
}
```

Функция `IntFilter()` принимает срез типа `[]int` и функцию-предикат, используемую для выявления элементов, которые требуется оставить. Она возвращает новый срез, содержащий только те элементы из исходного среза, для которых функция `predicate()` вернет `true`.

Фильтрация срезов – довольно распространенная задача, поэтому очень жаль, что `IntFilter()` может оперировать только срезами типа `[]int`. Но, к счастью, есть возможность создать обобщенную функцию фильтрации, используя тот же прием, что применялся при создании функции `SliceIndex()`.

```
func Filter(limit int, predicate func(int) bool, appender func(int)) {
    for i := 0; i < limit; i++ {
        if predicate(i) {
            appender(i)
        }
    }
}
```

Точно так же, как и функция `SliceIndex()`, функция `Filter()` ничего не знает о структуре данных, которыми оперирует, кроме указанного предела `limit`. В процессе фильтрации и конструирования результата функция `Filter()` опирается на функции `predicate()` и `appender()`.

```

readings := []int{4, -3, 2, -7, 8, 19, -11, 7, 18, -6}
even := make([]int, 0, len(readings))
Filter(len(readings), func(i int) bool { return readings[i]%2 == 0 },
    func(i int) { even = append(even, readings[i]) })
fmt.Println(even)
[4 2 8 18 -6]

```

Этот фрагмент выполняет ту же самую работу, что и фрагмент выше, только здесь новый срез `even` должен создаваться за пределами функции `Filter()`. Первая анонимная функция, передаваемая функции `Filter()`, является функцией-фильтром – она принимает индекс элемента среза и возвращает `true`, если элемент с этим индексом является четным числом. Вторая анонимная функция добавляет элемент оригинального среза с указанным индексом в новый срез. Оба среза должны находиться в области видимости, где создаются анонимные функции (замыкания), передаваемые функции `Filter()`, потому что анонимные функции должны захватить эти срезы, чтобы иметь возможность обращаться к ним.

```

parts := []string{"X15", "T14", "X23", "A41", "L19", "X57", "A63"}
var Xparts []string
Filter(len(parts), func(i int) bool { return parts[i][0] == 'X' },
    func(i int) { Xparts = append(Xparts, parts[i]) })
fmt.Println(Xparts)
[X15 X23 X57]

```

Этот второй пример просто подчеркивает независимость функции `Filter()` от типов данных, которыми она оперирует, – здесь она работает уже не с целыми числами, а со строками.

```

var product int64 = 1
Filter(26, func(i int) bool { return i%2 != 0 },
    func(i int) { product *= int64(i) })
fmt.Println(product)
7905853580625

```

Последний фрагмент иллюстрирует, что, подобно функции `SliceIndex()`, `Filter()` может использоваться не только для фильтрации срезов. Здесь функция `Filter()` применяется для вычисления произведения нечетных натуральных чисел в диапазоне `[1, 25]`.

5.6.7.1. Мемоизация истинных функций

Истинной функцией называется функция, всегда возвращающая один и тот же результат для одного и того же аргумента или аргументов и не имеющая побочных эффектов. Если истинная функция выполняет продолжительные вычисления и часто вызывается с одними и теми же аргументами, к ней можно применить прием мемоизации и тем самым увеличить ее производительность за счет увеличенного потребления памяти. Прием мемоизации основан на сохранении результатов вычислений, чтобы при последующих вызовах вместо выполнения дорогостоящих вычислений можно было просто вернуть результаты, полученные ранее.

Рекурсивный алгоритм вычисления чисел Фибоначчи является достаточно дорогостоящим и предусматривает многократное вычисление одних и тех же чисел, как видно на рис. 5.2. В данном случае простейший способ избежать этого – использовать нерекурсивный алгоритм, но, только чтобы показать, как пользоваться приемом мемоизации, попробуем создать рекурсивную, мемоизованную функцию вычисления чисел Фибоначчи.

```
type memoizeFunction func(int, ...int) interface{}
var Fibonacci memoizeFunction
func init() {
    Fibonacci = Memoize(func(x int, xs ...int) interface{} {
        if x < 2 {
            return x
        }
        return Fibonacci(x-1).(int) + Fibonacci(x-2).(int)
    })
}
```

Функция `Memoize()`, которая будет представлена чуть ниже, мемоизует любую функцию, принимающую хотя бы один целочисленный аргумент и возвращающую значение типа `interface{}`. Для удобства в примере был определен тип `memoizeFunction`, определяющий сигнатуру такой функции, и объявлена переменная `Fibonacci` для хранения ссылки на функцию этого типа. Затем в функции `init()` программы была создана анонимная функция, вычисляющая число Фибоначчи, и тут же передана функции `Memoize()`. Функция `Memoize()`, в свою очередь, возвращает функцию типа `memoizeFunction`, которая присваивается переменной `Fibonacci`.

В данном конкретном примере функция `Fibonacci` принимает только один аргумент, поэтому любые дополнительные целые числа просто игнорируются (то есть игнорируется аргумент `xs`, который в данном случае должен быть пустым срезом). Кроме того, при суммировании результатов рекурсивных вызовов необходимо использовать неконтролируемое приведение типов, чтобы преобразовать возвращаемые значения типа `interface{}` в их фактические значения типа `int`.

Теперь функцию `Fibonacci()` можно использовать как любую другую функцию, и благодаря мемоизации она не будет повторно вычислять числа, полученные ранее.

```
fmt.Println("Fibonacci(45) =", Fibonacci(45).(int))
Fibonacci(45) = 1134903170
```

Этот пример демонстрирует использование рекурсивной мемоизированной функции `Fibonacci()` и возвращаемый ею результат. Здесь для преобразования возвращаемого значения типа `interface{}` в значение типа `int` была использована операция неконтролируемого приведения типа. (Строго говоря, в этом не было необходимости, потому что функции вывода из пакета `fmt` способны выполнить такое преобразование автоматически, но здесь это было сделано с целью показать реалистичный способ использования.)

```
func Memoize(function memoizeFunction) memoizeFunction {
    cache := make(map[string]interface{})
    return func(x int, xs ...int) interface{} {
        key := fmt.Sprint(x)
        for _, i := range xs {
            key += fmt.Sprintf(",%d", i)
        }
        if value, found := cache[key]; found {
            return value
        }
        value := function(x, xs...)
        cache[key] = value
        return value
    }
}
```

Тут используется очень простая функция `Memoize()`. Она принимает в качестве аргумента функцию типа `memoizeFunction` (с

сигнатурой `func(int, ...int) interface{}`) и возвращает другую функцию с такой же сигнатурой.

Для хранения ранее вычисленных результатов здесь используется отображение со строковыми ключами и значениями типа `interface{}`. Отображение захватывается анонимной функцией (замыканием), которую возвращает функция `Memoize()`. Ключ конструируется путем перечисления через запятую значений всех целочисленных аргументов. (В качестве ключей отображений в языке Go могут использоваться только типы, полностью поддерживающие операторы `==` и `!=`, — строки соответствуют этому требованию, а срезы нет; §4.3.) После вычисления ключа проверяется, имеется ли в отображении соответствующая пара *ключ/значение*. Если имеется, повторные вычисления выполнять не требуется и можно просто вернуть сохраненное ранее значение. В противном случае производятся вычисления вызовом функции, переданной для мемоизации, с целочисленным аргументом или аргументами. Полученный результат сохраняется в отображении, чтобы впоследствии можно было не повторять вычисления с этим же набором аргументов, и полученное значение возвращается вызывающей программе.

Прием мемоизации отлично подходит для истинных функций (не важно, рекурсивных или нет), выполняющих продолжительные вычисления, которые часто вызываются с одним и тем же набором аргументов. Например, если в программе требуется много раз преобразовывать целые числа в римскую форму записи и при этом многие числа повторяются, тогда имеет смысл использовать функцию `Memoize()`, чтобы избежать повторяющихся вычислений. Естественно, чтобы определить необходимость мемоизации (или других приемов оптимизации), лучше всего провести хронометраж (например, с помощью пакета `time` или инструмента профилирования) операций.

```
var RomanForDecimal memoizeFunction
func init() {
    decimals := []int{1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1}
    romans := []string{"M", "CM", "D", "CD", "C", "XC", "L", "XL", "X",
        "IX", "V", "IV", "I"}
    RomanForDecimal = Memoize(func(x int, xs ...int) interface{} {
        if x < 0 || x > 3999 {
            panic("RomanForDecimal() only handles integers [0, 3999]")
        }
        var buffer bytes.Buffer
        for i, decimal := range decimals {
```



```
    remainder := x / decimal
    x %= decimal
    if remainder > 0 {
        buffer.WriteString(strings.Repeat(romans[i], remainder))
    }
}
return buffer.String()
})
}
```

Функция `RomanForDecimal()` объявляется глобально (в смысле глобально внутри пакета; см. главу 9) как функция типа `memoizeFunction` и создается в функции `init()`. Срезы `decimals` и `romans` являются локальными по отношению к функции `init()`, но они сохраняются в памяти так долго, пока существует функция `RomanForDecimal()`, сохраняющая эти срезы в замыкании.

Функции (и методы) в языке Go обладают огромными возможностями и поддерживают различные способы их обобщения, когда это необходимо.

5.7. Пример: сортировка с учетом отступов

В этом разделе рассматривается пользовательская функция, сортирующая срез строк. Особенностью этой функции (из-за которой ее нельзя заменить функцией `sort.Strings()` из стандартной библиотеки) является поддержка иерархической сортировки строк, то есть с учетом отступов. (Исходный текст функции можно найти в файле `indent_sort/indent_sort.go`.)

Имейте в виду, что алгоритм, использованный в функции `SortedIndentedStrings()`, основывается на одном важном допущении: величина отступа в строках определяется количеством начальных пробелов или символов табуляции, что позволяет работать в терминах байтов, а не многобайтных пробельных символов. (Если действительно потребуются обрабатывать многобайтные пробельные символы, самым простым решением будет заменить каждый такой символ пробелом или символом табуляции перед передачей строк функции `SortedIndentedStrings()`, например с помощью функции `strings.Map()`.)

Начнем с программы, вызывающей функцию и выводящей несоортированный и отсортированный срезы со строками в две колонки для сравнения.

Функция `SortedIndentedStrings()`, вспомогательные функции и типы используют рекурсивные функции, ссылки на функции и указатели на срезы. Поэтому, несмотря на возможность увидеть, что именно делает функция, реализация решения требует некоторых пояснений. Решение основывается на некоторых важных особенностях функций в языке Go, представленных в этой главе, а также на некоторых идеях и приемах, с которыми мы познакомились в главе 4 и которые более полно будут описаны в главе 6.

```
func main() {
    fmt.Println("|      Original      |      Sorted      |")
    fmt.Println("|-----|-----|")
    sorted := SortedIndentedStrings(original) // original - срез типа []string
    for i := range original {                // глобальная переменная
        fmt.Printf("|%-19s|%-19s|\n", original[i], sorted[i])
    }
}
```

Original	Sorted
-----	-----
Nonmetals	Alkali Metals
Hydrogen	Lithium
Carbon	Potassium
Nitrogen	Sodium
Oxygen	Inner Transitionals
Inner Transitionals	Actinides
Lanthanides	Curium
Europium	Plutonium
Cerium	Uranium
Actinides	Lanthanides
Uranium	Cerium
Plutonium	Europium
Curium	Nonmetals
Alkali Metals	Carbon
Lithium	Hydrogen
Sodium	Nitrogen
Potassium	Oxygen

Ключевыми элементами в этом решении являются типы `Entry` и `Entries`. Для каждой строки в исходном срезе создается значение типа `Entry`, поле `key` которого используется для сортировки, поле `value` предназначено для хранения оригинальной строки, а поле `children` — для хранения среза дочерних строк в виде значений `Entry`

(этот срез может быть пустым или хранить элементы типа `Entry`, также имеющие дочерние строки, и т. д.).

```

type Entry struct {
    key      string
    value    string
    children Entries
}

type Entries []Entry
func (entries Entries) Len() int { return len(entries) }
func (entries Entries) Less(i, j int) bool {
    return entries[i].key < entries[j].key
}
func (entries Entries) Swap(i, j int) {
    entries[i], entries[j] = entries[j], entries[i]
}

```

Интерфейс `sort.Interface`, объявленный в пакете `sort`, определяет три метода: `Len()`, `Less()` и `Swap()`, с теми же сигнатурами, что и одноименные методы типа `Entries`. Это означает, что значение типа `Entries` легко можно взять с помощью функции `sort.Sort()` из стандартной библиотеки.

```

func SortedIndentedStrings(slice []string) []string {
    entries := populateEntries(slice)
    return sortedEntries(entries)
}

```

Это экспортируемая (общедоступная) функция, выполняющая всю работу, точнее, она выполняет всю работу, вызывая вспомогательные функции. Функция `populateEntries()` принимает срез типа `[]string` и возвращает соответствующее ему значение типа `Entries` (основанного на типе `[]Entry`). Функция `sortedEntries()` принимает значение типа `Entries` и возвращает соответствующий ему срез типа `[]string` с иерархически (по отступам) отсортированными строками.

```

func populateEntries(slice []string) Entries {
    indent, indentSize := computeIndent(slice)
    entries := make(Entries, 0)
    for _, item := range slice {
        i, level := 0, 0
        for strings.HasPrefix(item[i:], indent) {

```

```

        i += indentSize
        level++
    }
    key := strings.ToLower(strings.TrimSpace(item))
    addEntry(level, key, item, &entries)
}
return entries
}

```

Функция `populateEntries()` начинается с определения в заданном срезе отступа на один уровень в виде строки (например, в виде строки из четырех пробелов) и его размера (количества байтов в отступе на один уровень). Затем она создает пустое значение типа `Entries` и начинает итерации по строкам в исходном срезе. Для каждой строки по величине отступа определяется ее уровень в иерархии и затем создается ключ. Далее она вызывает функцию `addEntry()` и передает ей уровень строки в иерархии, ключ, саму строку (`item`) и адрес среза `entries`, чтобы функция `addEntry()` смогла создать новое значение типа `Entry` и добавить его в срез `entries`. В конце срез `entries` возвращается вызывающей программе.

```

func computeIndent(slice []string) (string, int) {
    for _, item := range slice {
        if len(item) > 0 && (item[0] == ' ' || item[0] == '\t') {
            whitespace := rune(item[0])
            for i, char := range item[1:] {
                if char != whitespace {
                    return strings.Repeat(string(whitespace), i), i
                }
            }
        }
    }
    return "", 0
}

```

Эта функция определяет, какой символ (пробел или табуляция) используется для оформления отступов и как много символов применяется для оформления отступа на один уровень.

Теоретически эта функция должна выполнить итерации по всем строкам, потому что строки верхнего уровня не имеют отступов. Но, как только она обнаруживает первую строку, начинающуюся с символа пробела или табуляции, она возвращает строку, представляющую отступ на один уровень и количество символов в ней.

```
func addEntry(level int, key, value string, entries *Entries) {  
    if level == 0 {  
        *entries = append(*entries, Entry{key, value, make(Entries, 0)})  
    } else {  
        addEntry(level-1, key, value,  
            &((*entries)[entries.Len()-1].children))  
    }  
}
```

Эта рекурсивная функция создает новое значение типа `Entry` и добавляет его в срез `entries` либо непосредственно, либо как дочерний элемент в другое значение типа `Entry`, если строка `value` имеет отступ.

Срез `entries` должен передаваться функции по указателю типа `*Entries`, а не по ссылке (как по умолчанию передаются срезы), потому что здесь требуется добавить новый элемент в срез `entries`, тогда как операция добавления по ссылке может создать бесполезную локальную копию, оставив исходный срез нетронутым.

Если параметр `level` равен 0, значит, функции был передан элемент верхнего уровня, который следует добавить непосредственно в срез `*entries`. В действительности ситуация немного сложнее — уровень `level` определяется относительно текущего среза `*entries`, каковым первоначально является срез типа `Entries` верхнего уровня, но если функция была вызвана рекурсивно, ей будет передан другой срез типа `Entries`, принадлежащий дочернему элементу типа `Entry`. Новый элемент добавляется в срез с помощью встроенной функции `append()` и с использованием оператора разыменования `*` для доступа к значению `entries`. Это гарантирует, что любые изменения будут доступны вызывающей программе. Вновь добавленный элемент `Entry` получает указанные значения `key` и `value`, а также собственный срез типа `Entries` для хранения дочерних элементов. Это точка выхода из рекурсии.

Если значение `level` больше 0, значит, элемент содержит строку с отступом и его следует добавить как дочерний к первому предшествующему элементу с уровнем отступа на единицу меньше. Выполняется это с помощью рекурсивного вызова функции `addEntry()`, где последний параметр является, пожалуй, самым сложным выражением из всех встречавшихся до сих пор.

Подвыражение `entries.Len() - 1` возвращает целое число `int` — индекс последнего элемента в срезе `Entries`, на который указывает указатель `*entries`. (Метод `Entries.Len()` принимает значение типа

Entries, а не *Entries, но компилятор Go способен автоматически разыменовывать указатель entries и вызвать метод значения типа Entries, на которое он указывает.) Выражение целиком (кроме внешней части &(...)) ссылается на поле children последнего элемента Entry в срезе Entries (которое также имеет тип Entries). Таким образом, все выражение в целом возвращает адрес поля children последнего элемента Entry в срезе Entries – значение типа *Entries, то есть именно то, что требуется передать в рекурсивный вызов.

Чтобы помочь уяснить, что здесь происходит, ниже приводится менее компактный программный код, который выполняет то же самое, что и рекурсивный вызов функции addEntry() в блоке else.

```
theEntries := *entries
lastEntry := &theEntries[theEntries.Len()-1]
addEntry(level-1, key, value, &lastEntry.children)
```

В первой строке создается переменная theEntries для хранения значения, на которое ссылается указатель *entries. Это недорогая операция, так как фактически здесь ничего не копируется – в действительности переменная theEntries становится псевдонимом для доступа к значению Entries. Затем приобретается *адрес* (указатель) последнего элемента. Если не использовать оператор взятия адреса, выражение вернет новый элемент, являющийся копией последнего элемента, но копия здесь совершенно бесполезна. В заключение выполняется рекурсивный вызов функции addEntry(), которой передается адрес поля children последнего элемента.

```
func sortedEntries(entries Entries) []string {
    var indentedSlice []string
    sort.Sort(entries)
    for _, entry := range entries {
        populateIndentedStrings(entry, &indentedSlice)
    }
    return indentedSlice
}
```

В момент вызова функции sortedEntries() структура entries отражает содержимое колонки «Original» в выводе программы, где строки с отступами являются дочерними по отношению к своим родителям (как элементы в поле children типа Entries соответствующих родительских элементов).

После того как значение типа `Entries` будет заполнено, вызывает-ся функция `SortedIndentedStrings()` для создания соответствующего отсортированного среза типа `[]string`. Эта функция начинается с создания пустого среза типа `[]string` для сохранения результатов. Затем благодаря поддержке типом `Entries` методов интерфейса `sort.Interface` вызовом функции `sort.Sort()` выполняется сортировка элементов по значению поля `key`, потому что именно по этому полю выполняет сравнение метод `Entries.Less()`. Сортируются только элементы `Entry` верхнего уровня, а их дочерние элементы остаются неотсортированными.

Чтобы отсортировать дочерние элементы и их дочерние элементы рекурсивно, функция выполняет итерации по элементам верхнего уровня и для каждого вызывает функцию `populateIndentedStrings()`, которой передается элемент типа `Entry` и указатель на срез типа `[]string` для заполнения.

Срезы можно передавать функциям, если их требуется только изменить (например, заменить элемент), но здесь речь идет о добавлении новых элементов в срез. Встроенная функция `append()` иногда возвращает ссылку на новый срез (если оригинальный срез имеет недостаточную емкость). Эта проблема решается передачей указателя на срез и копированием ссылки на срез, возвращаемой функцией `append()`, по адресу, куда указывает указатель. (Если здесь не использовать указатель, мы просто получим локальный срез, недоступный вызывающей программе.) Альтернативное решение заключается в том, чтобы передавать сам срез и возвращать дополненный срез, который затем присваивать оригинальному срезу (например, `slice = function(slice)`), однако такое решение усложняет реализацию рекурсивных функций.

```
func populateIndentedStrings(entry Entry, indentedSlice *[]string) {
    *indentedSlice = append(*indentedSlice, entry.value)
    sort.Sort(entry.children)
    for _, child := range entry.children {
        populateIndentedStrings(child, indentedSlice)
    }
}
```

Эта функция добавляет значение поля `value` указанного элемента `entry` в конструируемый срез, сортирует его дочерние элементы и затем рекурсивно вызывает себя для каждого дочернего элемента.

В результате выполняется сортировка всех дочерних элементов в данном элементе `entry`, затем их дочерних элементов и т. д., таким образом весь срез `indentedSlice` оказывается отсортированным.

На этом заканчивается изучение встроенных типов данных и поддержки процедурного программирования в языке Go. В главе 6, с опорой на все это, будет рассказываться об особенностях объектно-ориентированного программирования, а в главе, следующей за ней, мы познакомимся с поддержкой параллельного программирования.

5.8. Упражнения

В этой главе предлагается выполнить четыре упражнения. Первое связано с изменением одного из примеров, представленных в главе, а другие требуют создания новых функций. Все упражнения очень короткие – первое и второе решаются просто, но над третьим и четвертым придется подумать!

1. Скопируйте содержимое каталога `archive_file_list` в каталог `my_archive_file_list`, например. Затем измените файл `archive_file_list/archive_file_list.go`: удалите весь программный код поддержки различных версий функции `ArchiveFileList()`, кроме `ArchiveFileListMap()`, которую нужно переименовать в `ArchiveFileList()`. Добавьте возможность обработки файлов с расширением `.tar.bz2` (тарболлы, сжатые утилитой `gzip`). Чтобы выполнить это упражнение, потребуется удалить примерно 11 строк из функции `main()`, удалить четыре функции, импортировать дополнительный пакет, добавить один элемент в отображение `FunctionForSuffix` и добавить несколько строк в функцию `TarFileList()`. Пример решения этого упражнения можно найти в файле `archive_file_list_ans/archive_file_list.go`.
2. Создайте нерекурсивные версии рекурсивных функций `IsPalindrome()`, показанных выше в этой главе (§5.6.4 и §5.6.5.2). Пример решения этого упражнения можно найти в файле `palindrome_ans/palindrome.go`, где нерекурсивная версия функции, поддерживающая только строки из ASCII-символов, занимает 10 строк и радикально отличается по структуре от рекурсивной версии. Нерекурсивная версия, поддерживающая строки с символами в кодировке UTF-8, занимает 14 строк и, напротив, очень похожа на рекурсивную версию, хотя и немного сложнее.

3. Создайте функцию `CommonPrefix()`, принимающую аргумент типа `[]string` и возвращающую строку с префиксом, общим для всех строк в исходном срезе (который может быть пустой строкой). Пример решения этого упражнения можно найти в файле `common_prefix/common_prefix.go`. Предложенное решение занимает 22 строки и использует для хранения строк срез типа `[]rune`, чтобы во время итераций можно было оперировать целыми символами, даже когда исходные строки содержат не ASCII-символы. В предложенном решении результат конструируется в буфере типа `bytes.Buffer`. Небольшой размер функции вовсе не означает, что написать ее будет просто! (Ниже, за следующим упражнением представлены некоторые примеры.)
4. Создайте функцию `CommonPathPrefix()`, принимающую срез типа `[]string` с путями в файловой системе и возвращающую строку с префиксом, общим для всех путей в исходном срезе (который может быть пустой строкой), – префикс должен содержать нуль или более полных компонентов пути. Пример решения этого упражнения можно найти в файле `common_prefix/common_prefix.go`. Предложенное решение занимает 27 строк и использует для хранения путей срез типа `[]string`, а также константу `filepath.Separator` для идентификации разделителя элементов путей на текущей платформе. В предложенном решении результат конструируется в срезе типа `[]string`, строки из которого в конце объединяются в единый путь вызовом функции `filepath.Join()`. Небольшой размер функции вовсе не означает, что написать ее будет просто! (Ниже представлены некоторые примеры.)

Далее приводится вывод программы `common_prefix`, где используются функции из упражнений 3 и 4. Первая строка в каждой паре содержит срез строк, а вторая – общий префикс и общий фрагмент пути, полученные с помощью функций `CommonPrefix()` и `CommonPathPrefix()`, с дополнительным признаком, указывающим, равен ли общий префикс общему фрагменту пути.

```
$ ./common_prefix
```

```
["/home/user/goeg" "/home/user/goeg/prefix" "/home/user/goeg/prefix/extra"]
char × path prefix: "/home/user/goeg" == "/home/user/goeg"
```

```
["/home/user/goeg" "/home/user/goeg/prefix" "/home/user/prefix/extra"]
char × path prefix: "/home/user/" != "/home/user"
```

```
["/pecan/π/goeg" "/pecan/π/goeg/prefix" "/pecan/π/prefix/extra"]  
char × path prefix: "/pecan/π/" != "/pecan/π"
```

```
["/pecan/π/circle" "/pecan/π/circle/prefix" "/pecan/π/circle/prefix/extra"]  
char × path prefix: "/pecan/π/circle" == "/pecan/π/circle"
```

```
["/home/user/goeg" "/home/users/goeg" "/home/userspace/goeg"]  
char × path prefix: "/home/user" != "/home"
```

```
["/home/user/goeg" "/tmp/user" "/var/log"]  
char × path prefix: "/" == "/"
```

```
["/home/mark/goeg" "/home/user/goeg"]  
char × path prefix: "/home/" != "/home"
```

```
["home/user/goeg" "/tmp/user" "/var/log"]  
char × path prefix: "" == ""
```



6. Объектно-ориентированное программирование

Цель этой главы – продемонстрировать особенности объектно-ориентированного программирования на языке Go. Программисты с опытом процедурного программирования (например, на языке C) обнаружат, что основы, на которых покоится все, о чем рассказывается в этой главе, уже знакомы им или рассматривались в предыдущих главах. А вот программисты с опытом объектно-ориентированного программирования на языках (таких как C++, Java, Python), использующих модель на основе наследования, должны будут оставить многие привычные им понятия и приемы, в частности имеющие отношение к наследованию, потому что в языке Go используется совершенно иной подход к объектно-ориентированному программированию.

Стандартная библиотека языка Go в большинстве своем предоставляет пакеты функций, тем не менее там, где это уместно, предоставляются собственные типы, имеющие методы. В предыдущих главах нам уже приходилось создавать значения некоторых таких типов (например, `regexp.Regexp` и `os.File`) и вызывать их методы. Кроме того, мы даже определяли свои собственные простые типы и соответствующие методы, например для поддержки вывода и сортировки, поэтому основы использования типов в языке Go и вызова их методов должны быть вам уже знакомы.

В первом, очень коротком разделе главы будут представлены некоторые ключевые понятия объектно-ориентированного программирования на языке Go. Во втором разделе будет рассматриваться порядок создания пользовательских типов без методов, где в отдельных подразделах будет показано, как добавлять методы к пользовательским типам, создавать функции-конструкторы и реализовывать проверку значений полей, – все основы, необходимые для создания полнофункциональных пользовательских типов. В третьем разделе рассматриваются интерфейсы, образующие фундамент для

безопасной динамической типизации в языке Go. Четвертый раздел охватывает структуры. Здесь будет представлено множество новых особенностей, о которых не рассказывалось в предыдущих главах.

В последнем длинном разделе главы будут представлены три законченных примера реализации пользовательских типов, при создании которых будут использоваться сведения из предыдущих разделов в этой главе и изрядный объем сведений из предыдущих глав. В первом примере будет реализован простой пользовательский тип с единственным значением, во втором – небольшое семейство типов и в третьем – обобщенный тип коллекций.

6.1. Ключевые понятия

Важнейшим отличием поддержки объектно-ориентированного программирования на языке Go от аналогичной поддержки в таких языках, как C++, Java и (в меньшей степени) Python, является отсутствие механизма наследования. Когда объектно-ориентированная модель программирования только начинала набирать популярность, возможность наследования рекламировалась как одно из самых больших преимуществ. Но теперь, спустя несколько десятилетий, оказалось, что эта особенность имеет ряд существенных недостатков, которые проявляются особенно сильно при сопровождении крупных систем. Вместо механизмов агрегирования и наследования, как в большинстве других объектно-ориентированных языков, Go поддерживает механизмы *агрегирования* (также называется композицией) и *встраивания*. Чтобы понять разницу между агрегированием и встраиванием, рассмотрим короткий фрагмент.

```
type ColoredPoint struct {  
    color.Color // Анонимное (безымянное) поле (встраивание)  
    x, y int     // Именованные поля (агрегирование)  
}
```

Здесь `color.Color` – это имя типа из пакета `image/color`, а `x` и `y` – имена полей типа `int`. В терминологии языка Go `color.Color`, `x` и `y` являются *полями* структуры `ColoredPoint`. Поле `color.Color` является анонимным (поскольку не имеет имени) и поэтому считается встроенным полем. Поля `x` и `y` – это именованные агрегированные поля. Если создать значение типа `ColoredPoint` (например, так: `point := ColoredPoint{}`), его поля будут доступны как `point.Color`, `point.x`

и `point.y`. Обратите внимание, что при обращении к полю типа из другого пакета указывается только последний компонент имени, то есть `Color`, а не `color.Color`. (Подробнее об этом будет рассказываться в §6.2.1.1, §6.3 и §6.4.)

Термины «класс», «объект» и «экземпляр», устоявшиеся в объектно-ориентированном программировании с поддержкой наследования, в языке Go вообще не используются. Вместо них используются термины «тип» и «значение», где значения пользовательских типов могут иметь методы.

В отсутствие наследования исчезли и *виртуальные функции*. Вместо этого механизма Go предлагает механизм *динамической типизации*. В языке Go *параметры* могут определяться как конкретные типы (такие как `int`, `string`, `*os.File`, `MyType`) или как интерфейсы, то есть значения, обладающие методами, которые определяются интерфейсами. В параметре, объявленном как интерфейс, можно передать любое значение, при условии что оно обладает методами этого интерфейса. Например, значение с методом, имеющим сигнатуру `Write([]byte) (int, error)`, можно передать как значение типа `io.Writer` (то есть как значение, отвечающее требованиям интерфейса `io.Writer`) любой функции, принимающей параметр типа `io.Writer`, независимо от фактического типа значения. Это обеспечивает невероятную гибкость и широчайшие возможности, особенно в соединении с поддержкой методов доступа встроенных значений в языке Go.

Одно из преимуществ механизма наследования состоит в том, что некоторые методы достаточно реализовать только один раз в базовом классе, и затем их можно использовать во всех его подклассах. То же самое в языке Go достигается двумя способами. Первый заключается в использовании механизма встраивания. При встраивании типов методы достаточно создать только один раз, во встраиваемом типе, и использовать их во всех других типах, включающих встраиваемый¹. Второй состоит в том, чтобы в каждом типе реализовать свои версии методов в виде тонких оберток (как правило, однострочных), которые просто передают работу единственной функции.

Другой необычной особенностью объектно-ориентированного программирования на языке Go является то обстоятельство, что интерфейсы, значения и методы определяются отдельно друг от друга.

¹ В некоторых других языках используемый в Go термин «встраивание» известен как «делегирование».

Интерфейсы используются для определения сигнатур методов, структуры – для определения агрегированных и встроенных значений. А *методы* – для реализации операций, поддерживаемых пользовательскими типами (которые часто являются структурами). В языке нет четкой связи между методами и каким-то определенным интерфейсом, но если тип обладает полным набором методов одного или нескольких интерфейсов, значения данного типа могут использоваться везде, где ожидаются эти интерфейсы. Любой тип соответствует пустому интерфейсу (`interface{}`), поэтому там, где ожидается пустой интерфейс, можно использовать значение любого типа.

В объектно-ориентированном программировании на языке Go отношения типа «является» (*is-a*) определяются интерфейсами исключительно в терминах сигнатур методов. Таким образом, значение, реализующее интерфейс `io.Reader` (то есть имеющее метод с сигнатурой `Read([]byte) (int, error)`), считается значением, обеспечивающим возможность чтения данных, не потому, что оно таковым является (например, файлом, буфером или значением какого-то другого типа), а потому, что имеет определенные методы, в данном случае `Read()`. Это иллюстрирует рис. 6.1 (ниже). Отношения типа «имеет» (*has-a*) определяются с помощью структур с агрегированными или встроенными значениями определенных типов, входящими в состав данного типа.

Встроенные типы не позволяют добавлять в них новые методы, однако на их основе легко можно создавать собственные типы и добавлять необходимые методы уже в них. Значения таких типов позволяют вызывать дополнительные методы и могут также использоваться в любых функциях, методах и с любыми операторами, поддерживающими базовый тип. Например, объявив тип `type Integer int`,

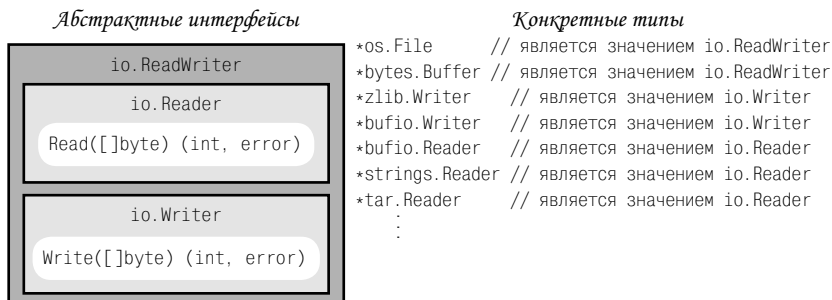


Рис. 6.1. Интерфейсы и типы для чтения и записи срезом байт

можно будет складывать значения этого типа как обычно, с помощью оператора `+`. Вдобавок к этому появляется возможность добавлять в него дополнительные методы, например `func (i Integer) Double() Integer { return i * 2 }`, как будет показано ниже (§6.2.1).

Пользовательские типы не только легко можно создавать на основе встроенных типов, они также весьма эффективны в использовании. Операции преобразования в/из пользовательских типов не имеют накладных расходов, потому что такие преобразования фактически выполняются на этапе компиляции. Это способствует «продвижению» значений встроенных типов в пользовательские типы, так как дает возможность использовать их методы, и «разжаловать» такие значения до встроенного типа перед передачей функциям с параметрами встроенных типов. Пример подобного «продвижения» был представлен ранее, когда значения типа `[]string` преобразовывались в значения типа `FoldedStrings` (§4.2.4), а первый пример «разжалования» будет представлен ниже в этой главе, на примере типа `Count` (§6.2.1).

6.2. Пользовательские типы

Пользовательские типы определяются с помощью ключевого слова `type`:

type *имяТипа* *определениеТипа*

Имя типа может быть любым допустимым идентификатором, уникальным в пределах пакета или функции. В качестве определения типа могут использоваться имя любого встроенного типа (например, `string`, `int`, срез, отображение или канал), интерфейса (§6.3), определение структуры (как было показано в предыдущих главах; дополнительная информация по этой теме приводится в §6.4) или сигнатура функции.

В некоторых ситуациях бывает достаточно просто определить пользовательский тип, в других бывает необходимо добавить методы, чтобы сделать его по-настоящему полезным. Ниже приводятся несколько примеров пользовательских типов без методов.

```
type Count int
type StringMap map[string]string
type FloatChan chan float64
```

Ни один из этих типов не выглядит особенно полезным, хотя их использование может улучшить удобочитаемость программы и в будущем изменить тип, лежащий в его основе. То есть они играют роль простейшего механизма абстракции.

```
var i Count = 7
i++
fmt.Println(i)
sm := make(StringMap)
sm["key1"] = "value1"
sm["key2"] = "value2"
fmt.Println(sm)
fc := make(FloatChan, 1)
fc <- 2.29558714939
fmt.Println(<-fc)
8
map[key2:value2 key1:value1]
2.29558714939
```

Типы, подобные типам `Count`, `StringMap` и `FloatChan`, основанные непосредственно на встроенных типах, можно использовать точно так же, как сами встроенные типы. Например, к значению пользовательского типа `StringSlice []string` можно применить встроенную функцию `append()`, но его необходимо будет преобразовать в значение встроенного типа (без накладных расходов, потому что эта операция выполняется на этапе компиляции), прежде чем передать функции, ожидающей получить значение этого встроенного типа. А иногда бывает необходимо выполнить обратное действие и преобразовать значение встроенного типа в значение пользовательского типа, чтобы получить возможность использовать дополнительные методы пользовательского типа. Пример такого преобразования был представлен ранее, когда в функции `SortFoldedStrings()` значения типа `[]string` преобразовывались в значения типа `FoldedStrings` (§4.2.4).

```
type RuneForRuneFunc func(rune) rune
```

При работе с *функциями высшего порядка* (§5.6.7) часто бывает удобно объявить тип, определяющий сигнатуру передаваемой функции. Здесь определяется сигнатура функции, принимающей и возвращающей единственное значение типа `rune`.

```
var removePunctuation RuneForRuneFunc
```

Переменная `removePunctuation` предназначена для хранения ссылки на функцию типа `RuneForRuneFunc` (то есть с сигнатурой `func(rune) rune`). Подобно всем переменным в языке Go, она автоматически инициализируется *нулевым значением*, то есть в данном случае значением `nil`.

```
phrases := []string{"Day; dusk, and night.", "All day long"}
removePunctuation = func(char rune) rune {
    if unicode.Is(unicode.Terminal_Punctuation, char) {
        return -1
    }
    return char
}
processPhrases(phrases, removePunctuation)
```

Здесь создается анонимная функция с сигнатурой, соответствующей `RuneForRuneFunc`, и передается пользовательской функции `processPhrases()`.

```
func processPhrases(phrases []string, function RuneForRuneFunc) {
    for _, phrase := range phrases {
        fmt.Println(strings.Map(function, phrase))
    }
}
Day dusk and night
All day long
```

Использование типа `RuneForRuneFunc` вместо `func(rune) rune` выглядит для человека более осмысленно. Кроме того, он образует дополнительный уровень абстракции. (Функция `strings.Map()` была описана в главе 3, §3.6.1.)

Создание пользовательских типов на основе встроенных типов или сигнатур функций может приносить определенную пользу, но она не очень велика. Добиться большего поможет добавление дополнительных методов, о чем рассказывается в следующем подразделе.

6.2.1. Добавление методов

Метод – это особого рода функция, которая вызывается относительно значения пользовательского типа, которое (обычно) передается вызываемому методу. Значение передается по указателю или по значению, в зависимости от того, как объявлен метод. Синтаксис определения

методов практически идентичен синтаксису определения функций, за исключением того, что между ключевым словом `func` и именем метода в круглых скобках указывается *приемник* (receiver) либо как имя типа, которому принадлежит метод, либо как имя переменной с типом. При вызове метода переменной-приемнику (если указана) автоматически присваивается значение или указатель на значение, от-носительно которого был произведен вызов метода.

В любой пользовательский тип можно добавить один или более методов. Приемником метода всегда будет значение типа или указатель на значение типа. Однако имена всех методов должны быть уникальными в пределах типа. Из требования к уникальности имени следует, что нельзя создать два метода с одинаковыми именами, один из которых принимает значение, а другой – указатель на значение. Еще одно следствие – отсутствие поддержки *перегруженных методов*, то есть методов с одинаковыми именами и разными сигнатурами. Один из способов реализовать подобие перегруженных методов заключается в создании методов с переменным числом аргументов (§5.6 и §5.6.1.2), однако в языке Go принято использовать функции с уникальными именами. Например, тип `strings.Reader` предоставляет три разных метода чтения: `strings.Reader.Read()`, `strings.Reader.ReadByte()` и `strings.Reader.ReadRune()`.

```

type Count int
func (count *Count) Increment() { *count++ }
func (count *Count) Decrement() { *count-- }
func (count Count) IsZero() bool { return count == 0 }

```

Это простой пользовательский тип, основанный на встроенном типе `int`, поддерживает три метода, из которых первые два принимают указатель на приемник, поскольку изменяют значение, относительно которого вызываются¹.

```

var count Count
i := int(count)
count.Increment()
j := int(count)
count.Decrement()
k := int(count)
fmt.Println(count, i, j, k, count.IsZero())
0 0 1 0 true

```

¹ В C++ и Java приемники всегда называются `this`, а в Python – `self`; в языке Go принято давать более значимые имена в зависимости от типа.

Данный фрагмент демонстрирует практическое применение типа `Count`. Пока этот тип не выглядит как нечто выдающееся, но он получит дальнейшее развитие в четвертом разделе этой главы.

Рассмотрим чуть более сложный пользовательский тип, на этот раз основанный на структуре. (Мы еще вернемся к этому примеру в §6.3.1.)

```
type Part struct {
    Id   int    // Именованное поле (агрегирование)
    Name string // Именованное поле (агрегирование)
}
func (part *Part) LowerCase() {
    part.Name = strings.ToLower(part.Name)
}
func (part *Part) UpperCase() {
    part.Name = strings.ToUpper(part.Name)
}
func (part Part) String() string {
    return fmt.Sprintf("%d %q", part.Id, part.Name)
}
func (part Part) HasPrefix(prefix string) bool {
    return strings.HasPrefix(part.Name, prefix)
}
```

Получение приемника по значению в методах `String()` и `HasPrefix()` реализовано, только чтобы показать, как это делается. Разумеется, методы, принимающие значение вместо указателя, не могут изменить значение, к которому применяются.

```
part := Part{5, "wrench"}
part.UpperCase()
part.Id += 11
fmt.Println(part, part.HasPrefix("w"))
"16 "WRENCH" false
```

Когда пользовательские типы основаны на структурах, их значения можно создавать, указывая имя типа с фигурными скобками за ним, содержащими список начальных значений полей. (И, как будет показано в следующем подразделе, в языке Go поддерживается возможность определять начальные значения лишь отдельных полей, а остальные автоматически будут инициализироваться нулевыми значениями.)

После создания значения `part` его можно использовать для вызова методов (например, `Part.UpperCase()`), обращаться к его экспортируемым (общедоступным) полям (например, `Part.Id`) и выводить его содержимое, полагаясь на функции вывода в языке Go, всегда использующие метод `String()` типа, если он имеется.

Множество методов типа – это множество всех методов, которые могут быть вызваны относительно значения этого типа.

Множество методов *указателя* на значение пользовательского типа включает в себя все методы этого типа, независимо от того, принимают ли они значение или указатель на него. Если относительно указателя вызвать метод, принимающий значение, компилятор Go автоматически разыменует указатель и в качестве приемника передаст методу значение.

Множество методов значения пользовательского типа включает в себя все методы этого типа, принимающие приемник по значению, – методы, принимающие указатель, не входят в это множество. Однако это не такое большое ограничение, как могло бы показаться, так как для вызова метода, принимающего указатель на приемник, достаточно просто вызвать этот метод относительно адресуемого значения (то есть относительно переменной, разыменованного указателя, элемента массива или среза, или адресуемого поля структуры). То есть вызов `value.Method()`, где `Method()` требует указатель на приемник, а `value` – адресуемое значение, компилятор Go будет интерпретировать как `(&value).Method()`.

Множество методов типа `*Count` включает три метода: `Increment()`, `Decrement()` и `IsZero()`, тогда как множество методов типа `Count` включает единственный метод – `IsZero()`. Все эти методы могут быть вызваны относительно указателя типа `*Count` и значения типа `Count`, если оно указано в адресуемой форме, как было показано выше. Множество методов типа `*Part` включает четыре метода: `LowerCase()`, `UpperCase()`, `String()` и `HasPrefix()`, а множество методов типа `Part` включает только методы `String()` и `HasPrefix()`. Однако методы `LowerCase()` и `UpperCase()` могут вызываться относительно адресуемых значений типа `Part`, как было показано выше.

Методы, принимающие приемник по значению, лучше подходят для небольших типов, таких как числа. подобные методы не могут изменить значение, относительно которого были вызваны, потому что в качестве приемника получают копию значения. Для значений больших типов или значений, которые должны изменяться в методе, следует предусматривать передачу приемника по указателю. Это

максимально уменьшит накладные расходы на вызов метода (так как приемник будет передаваться в виде 32- или 64-битного указателя, независимо от фактического размера значения).

6.2.1.1. Переопределение методов

Как будет показано далее в этой главе, составные типы на основе структур могут включать один или более типов (в том числе и интерфейсы) в виде *встраиваемых полей* (§6.4.1). Главное удобство такого подхода заключается в возможности вызывать методы встроенного типа относительно значения пользовательского типа, как если бы они были собственными методами этого типа, при этом в качестве приемников таким методам будут передаваться встроенные поля.

```
type Item struct {
    id      string // Именованное поле (агрегирование)
    price   float64 // Именованное поле (агрегирование)
    quantity int    // Именованное поле (агрегирование)
}

func (item *Item) Cost() float64 {
    return item.price * float64(item.quantity)
}

type SpecialItem struct {
    Item          // Анонимное поле (встраивание)
    catalogId int // Именованное поле (агрегирование)
}
```

Здесь тип `SpecialItem` встраивает тип `Item`. Это означает, что относительно значения типа `SpecialItem` можно вызвать метод `Cost()`, принадлежащий типу `Item`.

```
special := SpecialItem{Item{"Green", 3, 5}, 207}
fmt.Println(special.id, special.price, special.quantity, special.catalogId)
fmt.Println(special.Cost())
Green 3 5 207
15
```

Вызов `special.Cost()` компилятор Go будет интерпретировать как вызов метода `Item.Cost()`, потому что тип `SpecialItem` не имеет собственного метода `Cost()`, и передаст ему встроенное значение типа `Item`, а не все значение типа `SpecialItem`, относительно которого метод вызывался первоначально.

Как будет показано ниже, если имя какого-либо поля во встроенном типе `Item` совпадает с именем поля в типе `SpecialItem`, обратиться к полю во встроенном типе `Item` можно, указав имя типа как часть полного имени поля, например `special.Item.price`.

Методы встраиваемых полей можно *переопределять*, просто создавая для встраивающей структуры новые методы с теми же именами, что и методы встроенного поля. Например, предположим, что имеется следующий тип элемента:

```
type LuxuryItem struct {
    Item           // Анонимное поле(встраивание)
    markup float64 // Именованное поле (агрегирование)
}
```

Как отмечалось выше, если вызвать метод `Cost()` относительно значения типа `LuxuryItem`, будет вызван метод `Item.Cost()` встроенного типа, как и в случае со значением типа `SpecialItem`. Ниже приводятся три другие реализации, переопределяющие методы встраиваемого типа (разумеется, использоваться может только одна из них!).

```
/*
func (item *LuxuryItem) Cost() float64 { // Слишком подробная!
    return item.Item.price * float64(item.Item.quantity) * item.markup
}
func (item *LuxuryItem) Cost() float64 { // Ненужное дублирование операций!
    return item.price * float64(item.quantity) * item.markup
}
*/
func (item *LuxuryItem) Cost() float64 { // Идеальный вариант
    return item.Item.Cost() * item.markup
}
```

Последняя реализация использует уже имеющийся метод `Cost()`. Разумеется, переопределяющие методы не должны использовать методов встроенного типа, если в этом нет необходимости. (Поля, встраиваемые в структуры, описываются ниже, в §6.4.1.)

6.2.1.2. Методы-выражения

Методы-выражения, как и функции, можно передавать и присваивать переменным. *Метод-выражение* – это функция, принимающая в первом аргументе значение типа, которому данный метод

принадлежит. (В других языках программирования подобные методы часто называют *несвязанными методами*.)

```
asStringV := Part.String // Фактическая сигнатура: func(Part) string
sv := asStringV(part)
hasPrefix := Part.HasPrefix // Фактическая сигнатура: func(Part, string) bool
asStringP := (*Part).String // Фактическая сигнатура: func(*Part) string
sp := asStringP(&part)
lower := (*Part).LowerCase // Фактическая сигнатура: func(*Part)
lower(&part)
fmt.Println(sv, sp, hasPrefix(part, "w"), part)
"16 "WRENCH"" "16 "WRENCH"" true "16 "wrench""
```

Здесь создаются четыре метода-выражения: `asStringV()`, принимающий значение типа `Part` в единственном аргументе, `hasPrefix()`, принимающий значение типа `Part` в первом аргументе и строку – во втором, и методы `asStringP()` и `lower()`, оба принимающие указатель типа `*Part` в виде единственного аргумента.

Методы-выражения являются дополнительной особенностью, которая может пригодиться в необычных ситуациях.

Все пользовательские типы, созданные до сих пор, подвержены одному недостатку. Ни один из них не имеет механизма, гарантирующего допустимость данных, которыми инициализируются поля (или реализующего принудительную инициализацию допустимыми значениями), и ни один из них не имеет механизма, гарантирующего, что значению типа (или полям, если речь идет о составном типе) не будут присвоены недопустимые данные. Например, полям `Part.Id` и `Part.Name` могут быть присвоены любые значения. Но что, если потребуется наложить какие-либо ограничения? Например, чтобы поле `Part.Id` могло содержать только положительные целочисленные значения больше нуля или поле `Part.Name` могло содержать имена только в определенном формате? Решение этой проблемы описывается в следующем подразделе, где будет создан небольшой, но законченный пользовательский тип, автоматически осуществляющий проверку присваиваемых данных.

6.2.2. Типы с проверкой

Для многих простых пользовательских типов проверка просто не нужна. Например, в программе может иметься тип `Point { X, Y int }`, для которого любые значения полей `X` и `Y` будут допустимыми. Кроме

того, поскольку компилятор Go гарантирует инициализацию всех переменных (включая поля структур) соответствующими нулевыми значениями, это существенно снижает потребность в явных конструкторах.

В ситуациях, когда нулевые значения не подходят для инициализации, можно создать *функцию-конструктор*. Go не поддерживает неявного вызова конструкторов, поэтому функция-конструктор должна вызываться явно. Для этого следует описать тип как не допускающий инициализацию нулевым значением и реализовать одну или более функций-конструкторов для создания допустимых значений.

Аналогичный подход можно использовать для организации проверки значений полей. Такие поля можно сделать *неэкспортируемыми* (частными) и реализовать *методы доступа* к ним, в которых выполнять необходимые проверки¹.

Взгляните на следующий небольшой, но законченный тип, определение которого иллюстрирует все вышесказанное.

```
type Place struct {
    latitude, longitude float64
    Name                string
}

func New(latitude, longitude float64, name string) *Place {
    return &Place{saneAngle(0, latitude), saneAngle(0, longitude), name}
}

func (place *Place) Latitude() float64 { return place.latitude }
func (place *Place) SetLatitude(latitude float64) {
    place.latitude = saneAngle(place.latitude, latitude)
}

func (place *Place) Longitude() float64 { return place.longitude }
func (place *Place) SetLongitude(longitude float64) {
    place.longitude = saneAngle(place.longitude, longitude)
}

func (place *Place) String() string {
    return fmt.Sprintf("(%0.3f°, %0.3f°) %q", place.latitude,
        place.longitude, place.Name)
}

func (original *Place) Copy() *Place {
    return &Place{original.latitude, original.longitude, original.Name}
}
```

¹ В языке Go неэкспортируемыми (то есть видимыми только внутри пакета, где они объявлены) являются идентификаторы, начинающиеся с буквы в нижнем регистре, а экспортируемыми (видимыми в любом пакете, импортирующем данный пакет) являются идентификаторы, начинающиеся с буквы в верхнем регистре.

Тип `Place` является экспортируемым (за пределы пакета, где он определен), но его поля `latitude` и `longitude` – нет, потому что они требуют проверки. Здесь имеется функция-конструктор `New()`, гарантирующая создание допустимого значения типа `*place.Places`. В языке Go принято давать функциям-конструкторам имя `New()` или имена, начинающиеся с «New», если таких функций несколько. (Здесь не показана функция `saneAngle()`, потому что она не является методом описываемого типа – она принимает величину старого и нового угла и возвращает новое значение, если оно в заданном диапазоне, в противном случае возвращается старое значение угла.) А наличие методов чтения и записи неэкспортируемых полей гарантирует, что им могут быть присвоены только допустимые значения.

Наличие метода `String()` означает, что значения типа `*Place` полностью отвечают требованиям интерфейса `fmt.Stringer`, и обеспечивает вывод значений `*Place` в требуемом формате. Здесь также реализован метод `Copy()`, но в нем не выполняются никакие проверки, потому что известно, что копируемое значение уже является допустимым.

```
newYork := place.New(40.716667, -74, "New York") // newYork - значение типа *Place
fmt.Println(newYork)
baltimore := newYork.Copy() // Baltimore - значение типа *Place
baltimore.SetLatitude(newYork.Latitude() - 1.43333)
baltimore.SetLongitude(newYork.Longitude() - 2.61667)
baltimore.Name = "Baltimore"
fmt.Println(baltimore)
(40.717°, -74.000°) "New York"
(39.283°, -76.617°) "Baltimore"
```

Тип `Place` находится в пакете `place`, поэтому для создания значения типа `*Place` здесь вызывается функция `place.New()`. После создания значения типа `*Place` его методы могут вызываться так же, как методы любого другого типа из стандартной библиотеки.

6.3. Интерфейсы

Интерфейс – это тип, определяющий сигнатуры одного или более методов. Интерфейсы являются полностью абстрактными, поэтому нет никакой возможности создавать их экземпляры. Однако имеется возможность создавать переменные с типами интерфейсов, которым затем можно присваивать значения любого конкретного типа, обладающего методами интерфейса.

Тип `interface{}` – это интерфейс, определяющий пустое множество методов. Этому интерфейсу удовлетворяет любое значение, независимо от того, имеет оно методы или нет, – в конце концов, если значение имеет методы, это множество методов включает пустое множество наряду со множеством фактических методов. По этой причине тип `interface{}` можно использовать для представления *любых* значений. Относительно значения, переданного как значение типа `interface{}`, нельзя вызывать методы (даже если они действительно имеются), поскольку представляющий его интерфейс не имеет методов. То есть в общем случае лучше передавать значения либо с их фактическим типом, либо с интерфейсом, определяющим методы, которые предполагается использовать. Разумеется, доступ к методам значения, переданного как значение типа `interface{}`, можно получить, выполнив операцию приведения типа (§5.1.2), выбора по типу (§5.2.2.2) или прибегнув к помощи механизма интроспекции (§9.4.9).

Ниже приводится пример очень простого интерфейса.

```
type Exchanger interface {  
    Exchange()  
}
```

Интерфейс `Exchanger` определяет единственный метод `Exchange()`, не имеющий аргументов и ничего не возвращающий. Имя интерфейса здесь выбрано с учетом соглашений, принятых в языке Go, согласно которым имена интерфейсов должны заканчиваться на «er». Для интерфейсов вполне обычное явление – определять единственный метод. Например, интерфейсы `io.Reader` и `io.Writer` из стандартной библиотеки определяют по одному методу. Обратите внимание, что в действительности интерфейсы определяют API (Application Programming Interface – прикладной программный интерфейс), то есть нуль или более методов, но никак не регламентируют, что должны делать эти методы.

Непустой интерфейс сам по себе обычно не используется. Чтобы получить от него хоть какую-то пользу, необходимо создать несколько пользовательских типов с методами, определяемыми интерфейсом¹. Ниже приводятся два таких типа.

¹ При разработке фреймворка можно было бы определить несколько интерфейсов и потребовать от пользователей фреймворка самим создать типы, реализующие методы этих интерфейсов, для использования их с фреймворком.

```
type StringPair struct{ first, second string }
func (pair *StringPair) Exchange() {
    pair.first, pair.second = pair.second, pair.first
}
type Point [2]int
func (point *Point) Exchange() { point[0], point[1] = point[1], point[0] }
```

Типы `StringPair` и `Point` совершенно не похожи друг на друга, но, так как оба реализуют метод `Exchange()`, они оба удовлетворяют требованиям интерфейса `Exchanger`. Это означает, что можно создать значения типов `StringPair` и `Point` и передавать их функциям, принимающим значения типа `Exchanger`.

Обратите внимание: несмотря на то что оба типа, `StringPair` и `Point`, реализуют интерфейс `Exchanger`, это никак явно не отмечено в программном коде — здесь нет инструкций, таких как «implements» или «inherits». Простого факта, что типы `StringPair` и `Point` реализуют методы (в данном случае единственный метод) интерфейса, уже достаточно для компилятора Go, чтобы считать эти значения соответствующими данному интерфейсу.

Приемники передаются методам по *указателям* на значения их типов, поэтому методы могут изменять значения, относительно которых они вызываются.

Даже при том, что функции вывода в языке Go способны выводить значения пользовательских типов, обычно предпочтительнее бывает организовать более полный контроль над их строковым представлением. Этого легко можно добиться добавлением реализации метода, определяемого интерфейсом `fmt.Stringer`, то есть метода с сигнатурой `String() string`.

```
func (pair StringPair) String() string {
    return fmt.Sprintf("%q+%q", pair.first, pair.second)
}
```

Этот метод возвращает строку, содержащую значения строковых полей в двойных кавычках со знаком «+» между ними. При наличии такого метода функции вывода из пакета `fmt` будут использовать его для вывода значений типа `StringPair`, а также `*StringPairs`, потому что компилятор Go автоматически разыменовывает такие указатели, чтобы получить значение, на которое он указывает.

Следующий фрагмент демонстрирует создание нескольких значений типа `Exchanger`, несколько вызовов метода `Exchange()` и несколько

вызовов пользовательской функции `exchangeThese()`, принимающей значения типа `Exchanger`.

```

jekyll := StringPair{"Henry", "Jekyll"}
hyde := StringPair{"Edward", "Hyde"}
point := Point{5, -3}
fmt.Println("Before: ", jekyll, hyde, point)
jekyll.Exchange() // Интерпретируется как: (&jekyll).Exchange()
hyde.Exchange()   // Интерпретируется как: (&hyde).Exchange()
point.Exchange()  // Интерпретируется как: (&point).Exchange()
fmt.Println("After #1:", jekyll, hyde, point)
exchangeThese(&jekyll, &hyde, &point)
fmt.Println("After #2:", jekyll, hyde, point)
Before:  "Henry"+"Jekyll" "Edward"+"Hyde" [5 -3]
After #1: "Jekyll"+"Henry" "Hyde"+"Edward" [-3 5]
After #2: "Henry"+"Jekyll" "Edward"+"Hyde" [5 -3]

```

Все *переменные* создаются как значения, даже при том, что методы `Exchange()` требуют передачи приемника по указателю. Однако в этом нет никаких проблем, потому что, как отмечалось выше, компилятор Go автоматически передает адрес значения при вызове метода, требующего указатель, если значение, относительно которого производится вызов, является адресуемым. То есть вызов `jekyll.Exchange()` в примере выше автоматически интерпретируется как вызов `(&jekyll).Exchange()`. То же относится и к другим вызовам.

Функции `exchangeThese()` явно должны передаваться адреса значений. Если, к примеру, передать ей значение `hyde` типа `StringPair`, компилятор Go заметит, что тип `StringPairs` не соответствует интерфейсу `Exchanger`, поскольку он не определяет метод `Exchange()` с приемником типа `StringPair`, и прервет компиляцию с сообщением об ошибке. Однако, если передать значение типа `*StringPair` (например, `&hyde`), компиляция будет выполнена успешно. Это обусловлено тем, что, согласно объявлению, метод `Exchange()` ожидает получить приемник типа `*StringPair`, а это означает, что тип `*StringPairs` соответствует интерфейсу `Exchanger`.

Ниже приводится реализация функции `exchangeThese()`.

```

func exchangeThese(exchangers ...Exchanger) {
    for _, exchanger := range exchangers {
        exchanger.Exchange()
    }
}

```

Эта функция понятия не имеет, что ей передаются два указателя типа `*StringPairs` и один типа `*Point`. Единственное, что она требует, чтобы передаваемые ей параметры реализовали интерфейс `Exchanger`, — это требование проверяется компилятором, поэтому используемая здесь *динамическая типизация* безопасна по отношению к типам.

Помимо собственных пользовательских интерфейсов, значения могут также включать реализацию любых других интерфейсов, в том числе определяемых в стандартной библиотеке, как в случае с типом `StringPair`, реализующим метод `String()` интерфейса `fmt.Stringer`. Другой пример — интерфейс `io.Reader`, определяющий единственный метод с сигнатурой `Read([]byte) (int, error)`, который при вызове записывает байты данных из значения в указанный срез типа `[]byte`. Запись носит деструктивный характер, в том смысле что каждый записанный байт удаляется из значения, относительно которого выполнен был вызов.

```
func (pair *StringPair) Read(data []byte) (n int, err error) {
    if pair.first == "" && pair.second == "" {
        return 0, io.EOF
    }
    if pair.first != "" {
        n = copy(data, pair.first)
        pair.first = pair.first[n:]
    }
    if n < len(data) && pair.second != "" {
        m := copy(data[n:], pair.second)
        pair.second = pair.second[m:]
        n += m
    }
    return n, nil
}
```

Благодаря добавлению этого метода тип `StringPair` обеспечил поддержку интерфейса `io.Reader`. Поэтому теперь значения типа `StringPair` (точнее, значения типа `*StringPair`, потому что некоторые методы требуют передачи указателя на приемник) реализуют интерфейсы `Exchanger`, `fmt.Stringer` и `io.Reader`, при этом в программном коде не требуется явно указывать, что `*StringPair` «реализует» `Exchanger` или какие-то другие интерфейсы. И конечно же ничто не мешает добавить другие методы и тем самым обеспечить реализацию дополнительных интерфейсов.

Метод использует встроенную функцию `copy()` (§4.2.3). Эта функция может применяться для копирования данных из одного среза в другой срез того же типа, но здесь используется другая форма копирования – содержимого строки в срез типа `[]byte`. Функция `copy()` никогда не копирует больше байтов, чем может уместиться в целевом срезе `[]byte`, и возвращает количество скопированных байтов. Пользовательский метод `StringPair.Read()` записывает байты из строки `first` (и удаляет, которые были фактически записаны), и затем повторяет то же самое со строкой `second`. Если обе строки окажутся пустыми, метод вернет нуль и значение `io.EOF`. Метод с успехом справится со своей задачей, если блок во второй инструкции `if` будет выполняться безусловно и из третьей инструкции убрать второе условное выражение, хотя и ценой потери эффективности (возможно, значительной).

Передача приемника по указателю является совершенной необходимостью, потому что метод `Read()` изменяет значение, относительно которого вызывается. И вообще, при определении типа приемника предпочтение следует отдавать указателям, за исключением небольших по объему значений, потому что передача указателя дешевле передачи других значений, кроме самых маленьких по объему.

Создав метод `Read()`, его можно использовать, как показано ниже.

```
const size = 16
robert := &StringPair{"Robert L.", "Stevenson"}
david := StringPair{"David", "Balfour"}
for _, reader := range []io.Reader{robert, &david} {
    raw, err := ToBytes(reader, size)
    if err != nil {
        fmt.Println(err)
    }
    fmt.Printf("%q\n", raw)
}
"Robert L.Stevens"
"DavidBalfour"
```

В этом фрагменте создаются два значения, реализующие интерфейс `io.Reader`. Поскольку реализованный выше метод `StringPair.Read()` принимает указатель на приемник, требованиям интерфейса `io.Reader()` удовлетворяет значение типа `*StringPair`, но не удовлетворяет значение типа `StringPair`. В первом случае сначала было создано значение типа `StringPair`, а затем переменной `robert` For

был присвоен указатель на него, во втором случае переменной `david` было присвоено само значение типа `StringPair`, поэтому в срезе `[]io.Reader` пришлось использовать его адрес.

После создания переменных выполняются итерации по ним. К каждой из них применяется пользовательская функция `ToBytes()`, копирующая данные в срез типа `[]byte`, которые затем выводятся в виде строк в кавычках.

Функция `ToBytes()` принимает значение, реализующее интерфейс `io.Reader` (то есть *любое* значение, имеющее метод с сигнатурой `Read([]byte) (int, error)`, такое как `*os.File`), а также максимальное число копируемых байтов, и возвращает срез типа `[]byte`, содержащий данные из указанного значения, и признак ошибки.

```
func ToBytes(reader io.Reader, size int) ([]byte, error) {
    data := make([]byte, size)
    n, err := reader.Read(data)
    if err != nil {
        return data, err
    }
    return data[:n], nil // Отсечет все неиспользованные байты
}
```

Так же как функция `exchangeThese()`, представленная ранее, эта функция ничего не знает о конкретном типе переданного ей значения – ей достаточно, чтобы оно содержало реализацию интерфейса `io.Reader`.

В случае успешного чтения данных длина среза `data` уменьшается до количества фактически прочитанных байтов. Если этого не сделать, то при большом значении параметра `size` будет получен срез, где за прочитанными байтами следуют нулевые байты (со значением `0x00`). Например, без отсечения лишних байтов при чтении данных из значения `david` можно было бы получить вывод `"DavidBalfour\x00\x00\x00\x00"`.

Обратите внимание, что нет прямой связи между интерфейсом и какими-либо типами, реализующими этот интерфейс. В программном коде не требуется явно указывать, что некоторый тип «наследует», «расширяет» или «реализует» интерфейс, – достаточно просто реализовать в типе необходимые методы. Это делает язык Go невероятно гибким, он позволяет легко добавлять новые интерфейсы, типы и методы в любой момент, без опаски разрушить дерево наследования.

6.3.1. Встраивание интерфейсов

Интерфейсы (и структуры, как будет показано в следующем разделе) в языке Go обладают прекрасной поддержкой *встраивания*. Интерфейсы могут встраиваться в другие интерфейсы, что оказывает практически такой же эффект, как если бы сигнатуры методов во встраиваемом интерфейсе были объявлены в интерфейсе, куда он встраивается. Эту особенность иллюстрирует следующий простой пример.

```
type LowerCaser interface {
    LowerCase()
}
type UpperCaser interface {
    UpperCase()
}
type LowerUpperCaser interface {
    LowerCaser // Как если бы был объявлен метод LowerCase()
    UpperCaser // Как если бы был объявлен метод UpperCase()
}
```

Интерфейс `LowerCaser` определяет единственный метод `LowerCase()`, не имеющий аргументов и ничего не возвращающий. Интерфейс `UpperCaser` объявлен аналогично. Интерфейс `LowerUpperCaser` встраивает два других интерфейса. То есть, чтобы конкретный тип удовлетворял требованиям интерфейса `LowerUpperCaser`, он должен иметь методы `LowerCase()` и `UpperCase()`.

В этом коротком примере не видны все преимущества возможности встраивания интерфейсов. Однако, если в первые два интерфейса добавить дополнительные методы (например, `LowerCaseSpecial()` и `UpperCaseSpecial()`), интерфейс `LowerUpperCaser` автоматически включил бы их без изменения его определения.

```
type FixCaser interface {
    FixCase()
}
type ChangeCaser interface {
    LowerUpperCaser // Как если бы были объявлены методы LowerCase() и UpperCase()
    FixCaser         // Как если бы был объявлен метод FixCase()
}
```

Здесь были добавлены два новых интерфейса, и в результате получилась своеобразная иерархия встроенных интерфейсов, как показано на рис. 6.2.

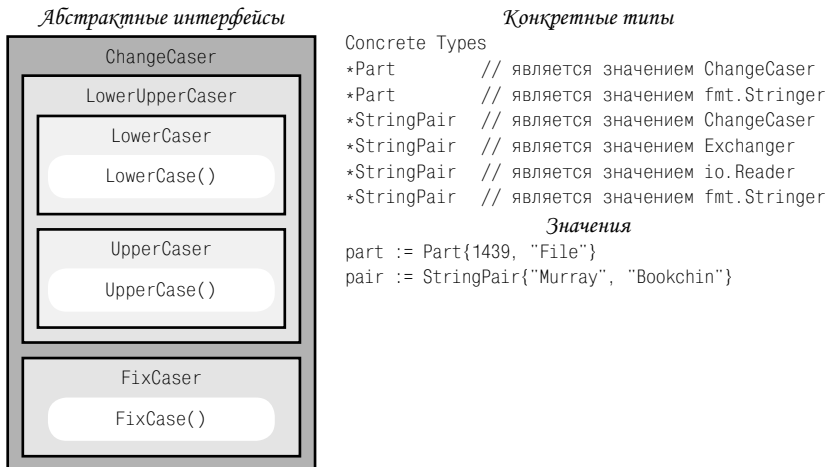


Рис. 6.2. Интерфейсы, типы и примеры значений

Интерфейсы сами по себе не используются – чтобы их задействовать, они должны быть реализованы в конкретных типах.

```

func (part *Part) FixCase() {
    part.Name = fixCase(part.Name)
}
        
```

Пользовательский тип `Part` уже был представлен выше. А здесь в него был добавлен дополнительный метод `FixCase()`, который воздействует на поле `Name`, подобно реализованным ранее методам `LowerCase()` и `UpperCase()`. Все методы, изменяющие регистр символов, требуют передачи приемника по указателю, потому что изменяют само значение, относительно которого вызываются. Методы `LowerCase()` и `UpperCase()` реализованы с использованием функций из стандартной библиотеки, а метод `FixCase()` опирается на пользовательскую функцию `fixCase()` – такой прием, когда маленькие методы выполняют свою работу с помощью функций, широко используется при программировании на языке Go.

Метод `Part.String()` (выше) реализует интерфейс `fmt.Stringer` из стандартной библиотеки и гарантирует, что все значения типа `Part` (или `*Part`) будут выводиться с использованием строки, возвращаемой этим методом.

```
func fixCase(s string) string {
    var chars []rune
    upper := true
    for _, char := range s {
        if upper {
            char = unicode.ToUpper(char)
        } else {
            char = unicode.ToLower(char)
        }
        chars = append(chars, char)
        upper = unicode.IsSpace(char) || unicode.Is(unicode.Hyphen, char)
    }
    return string(chars)
}
```

Эта простая функция возвращает копию переданной ей строки, где все символы преобразованы в нижний регистр, кроме самого первого и первых символов после пробелов или дефисов, которые преобразуются в верхний регистр. Например, для исходной строки "lobelia sackville-baggins" функция вернет "Lobelia Sackville-Baggins".

Естественно, поддержку всех представленных выше интерфейсов можно реализовать в любом пользовательском типе.

```
func (pair *StringPair) UpperCase() {
    pair.first = strings.ToUpper(pair.first)
    pair.second = strings.ToUpper(pair.second)
}
func (pair *StringPair) FixCase() {
    pair.first = fixCase(pair.first)
    pair.second = fixCase(pair.second)
}
```

Здесь в тип `StringPair`, созданный ранее, были добавлены методы, реализующие интерфейсы `LowerCaser`, `UpperCaser` и `FixCaser`, — метод `StringPair.LowerCase()` не показан, потому что он по своей структуре идентичен методу `StringPair.UpperCase()`.

Оба типа, `*Part` и `*StringPair`, удовлетворяют требованиям всех рассматриваемых интерфейсов, включая интерфейс `ChangeCaser`, потому что он встраивает интерфейсы, уже реализуемые типом. Оба они также реализуют интерфейс `fmt.Stringer` из стандартной библиотеки. А тип `*StringPair` дополнительно реализует пользовательский интерфейс `Exchanger` и интерфейс `io.Reader` из стандартной библиотеки.

Нет необходимости включать реализацию всех интерфейсов, например если отказаться от метода `StringPair.FixCase()`, тогда тип `*StringPair` удовлетворял бы требованиям только интерфейсов `LowerCaser`, `UpperCaser`, `LowerUpperCaser`, `Exchanger`, `fmt.Stringer` и `io.Reader`.

Создадим пару этих значений и посмотрим, как можно использовать их методы.

```
toastRack := Part{8427, "TOAST RACK"}
toastRack.LowerCase()
lobelia := StringPair{"LOBELIA", "SACKVILLE-BAGGINS"}
lobelia.FixCase()
fmt.Println(toastRack, lobelia)
"8427 "toast rack"" "Lobelia"+"Sackville-Baggins"
```

Методы вызываются и ведут себя, как и ожидалось. Но что, если имеется множество таких значений и требуется вызвать метод для каждого из них? Ниже демонстрируется не самое лучшее решение этой задачи.

```
for _, x := range []interface{}{&toastRack, &lobelia} { // НЕБЕЗОПАСНО!
    x.(LowerUpperCaser).UpperCase() // Неконтролируемое приведение типа
}
```

Здесь необходимо использовать указатели на значения, потому что все методы, изменяющие регистр символов, изменяют само значение, относительно которого вызываются, и потому требуют передачи приемников по указателю.

Подход, представленный в этом фрагменте, имеет два недостатка. Меньший состоит в том, что в операции неконтролируемого приведения типа используется интерфейс `LowerUpperCaser`, более обобщенный, чем здесь требуется. Было бы хуже, если бы здесь использовался еще более обобщенный интерфейс `ChangeCaser`. Но здесь нельзя использовать интерфейс `FixCaser`, так как он определяет только метод `FixCase()`. Лучше всего тут использовать минимально необходимый интерфейс, в данном случае интерфейс `UpperCaser`. Самый большой недостаток заключается в использовании именно неконтролируемого приведения типов, поскольку эта операция может возбудить аварийную ситуацию!

```
for _, x := range []interface{}{&toastRack, &lobelia} {
    if x, ok := x.(LowerCaser); ok { // затеняющая переменная
        x.LowerCase()
    }
}
```

В этом фрагменте реализовано более безопасное решение и используется самый специализированный интерфейс, но он выглядит достаточно громоздко. Проблема – в использовании среза со значениями универсального типа `interface{}`, а не определенного типа или типов, реализующих определенный интерфейс. Конечно, если все, что доступно в данной точке программы, – это срез типа `[] interface{}`, тогда данное решение – лучшее, что можно сделать.

```
for _, x := range []FixCaser{&toastRack, &lobelia} { // Идеально
    x.FixCase()
}
```

Этот фрагмент иллюстрирует самое удачное решение: вместо операции приведения типа для значений универсального типа `interface{}` здесь срез определен как срез со значениями типа `FixCasers` – наиболее специализированного интерфейса, достаточного для решения поставленной задачи, а все проверки типов перекладываются на компилятор.

Еще одна гибкая особенность интерфейсов состоит в том, что они могут создаваться по факту. Например, допустим, что в программе было создано несколько пользовательских типов, часть из которых имеет метод `IsValid() bool`. Если позднее обнаружится, что нужна некоторая функция, принимающая значение одного из пользовательских типов и вызывающая метод `IsValid()`, если он поддерживается значением, этого легко можно было бы добиться, как показано ниже.

```
type IsValider interface {
    IsValid() bool
}
```

Сначала нужно объявить интерфейс, определяющий методы, наличие которых требуется проверить.

```
if thing, ok := x.(IsValider); ok {
    if !thing.IsValid() {
        reportInvalid(thing)
    } else {
        // ... обработать допустимое значение ...
    }
}
```

Имея интерфейс, можно легко проверить любое значение на наличие метода `IsValid()` `bool` и вызвать его.

Интерфейсы обеспечивают очень мощный механизм абстракции, позволяющий определять множества методов так, чтобы в функциях и методах можно было использовать параметры с типом интерфейса, для которых интерес представляет лишь набор операций, которые они поддерживают, а не их конкретные типы. Ниже в этой главе будут представлены дополнительные примеры использования интерфейсов (§6.5.2).

6.4. Структуры

Простейшие пользовательские типы в программах на языке Go основываются на встроенных типах, например инструкция `type Integer int` создаст пользовательский тип `Integer`, к которому можно будет добавить собственные методы. Пользовательские типы могут также создаваться на основе структур, используемых для агрегирования и встраивания значений. Это особенно удобно, когда эти значения, называемые *полями*, имеют разные типы и не могут храниться в общем срезе (если только не использовать срез типа `[]interface{}`). Структуры в языке Go ближе к структурам в языке C, чем в C++ (например, они не являются классами), но более удобны в использовании благодаря отличной поддержке встраивания.

В предыдущих главах и в этой главе приводилось немало примеров структур, и еще много их встретится в оставшейся части книги. Тем не менее некоторые особенности структур еще не были описаны, поэтому для начала рассмотрим несколько примеров, демонстрирующих их.

```
points := [][]int{{4, 6}, {}, {-7, 11}, {15, 17}, {14, -8}}
for _, point := range points {
    fmt.Printf("(%d, %d) ", point[0], point[1])
}
```

Переменная `points` во фрагменте выше – это срез с массивами типа `[2]int`, поэтому для получения каждой координаты приходится использовать оператор индексирования `[]`. (Элемент `{}` – это то же самое, что и `{0, 0}`, благодаря автоматической инициализации переменных в языке Go.) Такой подход вполне пригоден для небольших объемов простых данных, но существует более удобный способ реализовать то же самое – с помощью анонимной структуры.

```
points := []struct{ x, y int }{{4, 6}, {}, {-7, 11}, {15, 17}, {14, -8}}
for _, point := range points {
    fmt.Printf("(%d, %d) ", point.x, point.y)
}
```

Переменная `points` в данном — это срез со структурами `struct{ x, y int}`. Несмотря на то что сама структура не имеет имени, в программном коде можно обращаться к ее полям по их именам, что гораздо проще, чем использовать массивы и индексы.

6.4.1. Структуры: агрегирование и встраивание

Структуры поддерживают такую же возможность *встраивания*, как интерфейсы или другие типы, то есть они могут включаться в другие структуры в виде поля, состоящего только из имени структуры. (Разумеется, если такому полю дать имя, получится *агрегированное* именованное поле, а не встроенное анонимное.)

Поля встроенного поля обычно доступны непосредственно, с помощью оператора точки (`.`), без упоминания имени типа, но если вмещающая структура имеет поле с тем же именем, что и поле во встроенной структуре, тогда необходимо использовать имя типа встроенной структуры, чтобы устранить неоднозначность.

Имена полей в структуре должны быть уникальными. К встроенным (то есть анонимным) полям предъявляются особенно строгие требования уникальности. Например, если в структуре имеется встроенное поле типа `Integer`, в нее также можно добавить другие поля с именами, например `Integer2` или `BigInteger`, поскольку они отчетливо различимы. Но в эту структуру уже нельзя будет добавить поле с именем, например, `Matrix.Integer` или `*Integer`, потому что уникальность полей определяется по последним компонентам в их именах, а последние компоненты в них совпадают с именем встроенного поля `Integer`.

6.4.1.1. Встраивание значений

Начнем с изучения простого примера двух структур.

```
type Person struct {
    Title    string // Именованное поле(агрегирование)
    Forenames []string // Именованное поле(агрегирование)
    Surname  string  // Именованное поле(агрегирование)
```

```

}
type Author1 struct {
    Names    Person // Именованное поле(агрегирование)
    Title    []string // Именованное поле(агрегирование)
    YearBorn int     // Именованное поле(агрегирование)
}

```

В предыдущих главах демонстрировалось множество похожих примеров. Здесь все поля в структуре Author1 имеют имена. Ниже показано, как можно использовать эти структуры и что выводится (благодаря наличию метода Author1.String(), который здесь не показан).

```

author1 := Author1{Person{"Mr", []string{"Robert", "Louis", "Balfour"},
    "Stevenson"}, []string{"Kidnapped", "Treasure Island"}, 1850}
fmt.Println(author1)
author1.Names.Title = ""
author1.Names.Forenames = []string{"Oscar", "Fingal", "O'Flahertie",
    "Wills"}
author1.Names.Surname = "Wilde"
author1.Title = []string{"The Picture of Dorian Gray"}
author1.YearBorn += 4
fmt.Println(author1)
Stevenson, Robert Louis Balfour, Mr (1850) "Kidnapped" "Treasure Island"
Wilde, Oscar Fingal O'Flahertie Wills (1854) "The Picture of Dorian Gray"

```

Здесь сначала создается значение типа Author1, затем по порядку заполняются все его поля, и, наконец, это значение выводится. Далее поля структуры изменяются, и она снова выводится в консоль.

```

type Author2 struct {
    Person      // Анонимное поле(встраивание)
    Title       []string // Именованное поле(агрегирование)
    YearBorn int  // Именованное поле(агрегирование)
}

```

Для встраивания анонимного поля используется имя типа (или интерфейса, как будет показано ниже) без определения имени поля. Обращаться к полям в этом поле можно непосредственно (то есть без указания имени типа или интерфейса) или используя имя типа или интерфейса, если это необходимо для устранения неоднозначности, при наличии одноименных полей во вмещающей структуре.

Структура `Author2` встраивает структуру `Person` как анонимное поле. Это означает, что к полям структуры `Person` можно обращаться непосредственно (за исключением случаев, когда необходимо избежать неоднозначности).

```
author2 := Author2{Person{"Mr", []string{"Robert", "Louis", "Balfour"},
    "Stevenson"}, []string{"Kidnapped", "Treasure Island"}, 1850}
fmt.Println(author2)
author2.Title = []string{"The Picture of Dorian Gray"}
author2.Person.Title = "" // Имя типа используется для устранения неоднозначности
author2.Forenames = []string{"Oscar", "Fingal", "O'Flahertie", "Wills"}
author2.Surname = "Wilde" // То же, что и: author2.Person.Surname = "Wilde"
author2.YearBorn += 4
fmt.Println(author2)
```

В этом фрагменте повторяется программный код, где использовалась структура `Author1`, но на этот раз в нем используется структура `Author2`. Он производит идентичный вывод (предполагается наличие метода `Author2.String()`, возвращающего то же строковое значение, что и метод `Author1.String()`).

За счет встраивания структуры `Person` в виде анонимного поля удалось получить практически тот же эффект, как при непосредственном добавлении полей структуры `Person`, но не совсем, потому что при непосредственном добавлении полей в окончательной структуре получилось бы два поля `Title`, что привело бы к ошибке во время компиляции.

Значение типа `Author2` создается точно так же, как значение типа `Author1`, но на этот раз к полям структуры `Person` можно обращаться непосредственно (например, `author2.Forenames`), за исключением случаев, когда требуется избежать неоднозначности (`author2.Person.Title` вместо `author2.Title`).

6.4.1.2. Встраивание анонимных значений с методами

Если встроенное поле обладает *методами*, их можно вызывать относительно вмещающей структуры, но в качестве *приемников* таким методам будет передаваться анонимное поле.

```
type Tasks struct {
    slice []string // Именованное поле (агрегирование)
    Count          // Анонимное поле (встраивание)
```



```
}  
func (tasks *Tasks) Add(task string) {  
    tasks.slice = append(tasks.slice, task)  
    tasks.Increment() // то же, что и: tasks.Count.Increment()  
}  
func (tasks *Tasks) Tally() int {  
    return int(tasks.Count)  
}
```

Определение типа `Count` было показано выше (§6.2.1). Структура `Tasks` имеет два поля: агрегированный срез со строками и встроенное значение `Count`. Как видно в реализации метода `Tasks.Add()`, к методам анонимного поля типа `Count` можно обращаться непосредственно.

```
tasks := Tasks{}  
fmt.Println(tasks.IsZero(), tasks.Tally(), tasks)  
tasks.Add("One")  
tasks.Add("Two")  
fmt.Println(tasks.IsZero(), tasks.Tally(), tasks)  
true 0 {[ ] 0}  
false 2 {[One Two] 2}
```

Здесь создается значение типа `Tasks` и вызываются его методы `Tasks.Add()`, `Tasks.Tally()` и `Tasks.Count.IsZero()` (как `Tasks.IsZero()`). Даже при том, что этот тип не имеет собственного метода `Tasks.String()`, функции вывода, имеющиеся в языке Go, воспроизводят вполне понятный вывод при попытке вывести значение `Tasks`. (Обратите внимание, что вызов метода `Count()` для структуры `Tasks`, если бы он имелся, не был бы скомпилирован, потому что это вызвало бы конфликт имен со встроенным значением `Tasks.Count`.)

При вызове метода встроенного поля важно помнить, что в качестве приемника методу передается само встроенное поле. То есть при вызове `Tasks.IsZero()`, `Tasks.Increment()` и других методов типа `Count` относительно значения типа `Tasks` этим методам передается значение типа `Count` (или `*Count`), а не `Tasks`.

В этом примере тип `Tasks` имеет собственные методы (`Add()` и `Tally()`), а также методы встроенного поля типа `Count` (`Increment()`, `Decrement()` и `IsZero()`). Конечно, в типе `Tasks` можно переопределить любые или даже все методы типа `Count`, просто реализовав методы с теми же именами. (Подобный пример был представлен выше, §6.2.1.1.)

6.4.1.3. Встраивание интерфейсов

Помимо агрегирования и встраивания в структуры конкретных типов, имеется также возможность агрегирования и встраивания интерфейсов. (Естественно, обратное – агрегирование и встраивание структур в интерфейсы – невозможно, потому что интерфейс является полностью абстрактным типом, то есть такое агрегирование и встраивание является сущей бессмыслицей.) Когда структура включает агрегированное (именованное) или встроенное (анонимное) поле с типом интерфейса, это означает, что структура может хранить в таком поле любое значение, реализующее методы указанного интерфейса.

Закончим обсуждение структур простым примером, демонстрирующим, как можно организовать поддержку параметров командной строки, имеющих короткие и длинные имена (например, «-o» и «--outfile») и принимающих значения определенных типов (`int`, `float64`, `string`), а также некоторых общих методов. (Этот пример создавался в учебных целях, а потому не блещет элегантностью. Полнофункциональный инструмент для работы с параметрами командной строки можно найти в стандартной библиотеке, в виде пакета `flag`, или в виде сторонних пакетов на сайте godashboard.appspot.com/project.)

```
type Optioner interface {
    Name() string
    IsValid() bool
}
type OptionCommon struct {
    ShortName string "short option name"
    LongName  string "long option name"
}
```

Интерфейс `Optioner` определяет общие методы для всех типов параметров, поддержку которых требуется реализовать. Структура `OptionCommon` имеет два поля, общих для всех параметров. В языке Go имеется возможность снабжать поля структур строковыми примечаниями (в терминологии Go они называются *тегами*). Эти теги не имеют функционального назначения, но, в отличие от комментариев, они доступны с применением механизма рефлексии (§9.4.9). Некоторые программисты используют теги для нужд проверки допустимости значений полей, например определив тег `"check:len(2,30)"` для строки или тег `"check:range(0,500)"` для числа и дополнительную семантику в программе.

```
type IntOption struct {
    OptionCommon // Анонимное поле (встраивание)
    Value, Min, Max int // именованное поле (агрегирование)
}
func (option IntOption) Name() string {
    return name(option.ShortName, option.LongName)
}
func (option IntOption) IsValid() bool {
    return option.Min <= option.Value && option.Value <= option.Max
}
func name(shortName, longName string) string {
    if longName == "" {
        return shortName
    }
    return longName
}
```

Это законченная реализация типа `IntOption` и неэкспортируемой функции `name()`. Благодаря встраиванию структуры `OptionCommon` появляется возможность обращаться к ее полям непосредственно, как это делается в методе `IntOption.Name()`. Структура `IntOption` удовлетворяет требованиям интерфейса `Optioner` (поскольку предоставляет методы `Name()` и `IsValid()` с требуемыми сигнатурами).

Несмотря на простоту функции `name()`, было принято реализовать обработку именно в отдельной функции, а не внутри метода `IntOption.Name()`. Это делает метод `IntOption.Name()` очень коротким и дает возможность повторно использовать данную функцию в других типах параметров. Так, например, тела методов `GenericOption.Name()` и `StringOption.Name()` совершенно идентичны телу метода `IntOption.Name()` и состоят из единственной инструкции, при этом все три метода для выполнения фактической работы вызывают функцию `name()`. Это типичный шаблон программирования на языке Go, и он снова будет использован в последнем разделе этой главы.

Реализация структуры `StringOption` очень похожа на реализацию структуры `IntOption`, поэтому здесь она не показана. (Различия заключаются только в поле `Value`, имеющем тип `string`, и в методе `IsValid()`, возвращающем `true`, если поле `Value` содержит непустое значение.) При реализации структуры `FloatOption` был использован прием встраивания интерфейса, только чтобы показать, как это делается.

```
type FloatOption struct {
    Optioner      // Анонимное поле (встраивание интерфейса: требует конкретный тип)
    Value float64 // Именованное поле (агрегирование)
}
```

Это законченная реализация структуры `FloatOption`. Встроенное поле `Optioner` означает, что при создании значений типа `FloatOption` встроенному полю должно присваиваться значение, удовлетворяющее требованиям интерфейса `Optioner`.

```
type GenericOption struct {
    OptionCommon // Анонимное поле (встраивание)
}
func (option GenericOption) Name() string {
    return name(option.ShortName, option.LongName)
}
func (option GenericOption) IsValid() bool {
    return true
}
```

Это законченная реализация структуры `GenericOption` — типа, удовлетворяющего требованиям интерфейса `Optioner`.

Тип `FloatOption` имеет встроенное поле типа `Optioner`, поэтому для него требуется указывать значение конкретного типа, реализующего интерфейс `Optioner`. Это требование можно удовлетворить присваиванием полю `Optioner` в структуре `FloatOption` значения типа `GenericOption`.

Теперь, когда в наличии имеются все фрагменты мозаики (`IntOption`, `FloatOption` и др.), посмотрим, как создавать и использовать значения этих типов.

```
fileOption := StringOption{OptionCommon{"f", "file"}, "index.html"}
topOption := IntOption{
    OptionCommon: OptionCommon{"t", "top"},
    Max: 100,
}
sizeOption := FloatOption{
    GenericOption{OptionCommon{"s", "size"}, 19.5}
for _, option := range []Optioner{topOption, fileOption, sizeOption} {
    fmt.Print("name=", option.Name(), " • valid=", option.IsValid())
    fmt.Print(" • value=")
    switch option := option.(type) { // затеняющая переменная
```

```

case IntOption:
    fmt.Print(option.Value, " • min=", option.Min,
        " • max=", option.Max, "\n")
case StringOption:
    fmt.Println(option.Value)
case FloatOption:
    fmt.Println(option.Value)
    }
}
name=top • valid=true • value=0 • min=0 • max=100
name=file • valid=true • value=index.html
name=size • valid=true • value=19.5

```

Вначале создается переменная `fileOption` типа `StringOption`, каждому полю которой присваивается соответствующее значение. Но в переменной `topOption` типа `IntOption` присвоить значение необходимо только полям `OptionCommon` и `Max`, потому что для других полей (`Value` и `Min`) прекрасно подходят нулевые значения. В языке Go допускается создавать структуры и инициализировать лишь отдельные поля, используя синтаксис `имяПоля: значениеПоля`. В этом случае всем полям, не инициализированным явно, автоматически присваиваются соответствующие нулевые значения.

Первым полем в значении `sizeOption` типа `FloatOption` является интерфейс `Optioner`, поэтому ему следует присвоить значение конкретного типа, удовлетворяющего требованиям этого интерфейса. Здесь для этой цели создано значение типа `GenericOption`.

Создав три разных параметра, по ним можно выполнить итерации, используя срез типа `[]Optioner`, то есть срез со значениями, удовлетворяющими требованиям интерфейса `Optioner`. Внутри цикла переменной `option` по очереди присваивается каждый параметр (типа `Optioner`). Относительно переменной `option` можно вызывать любые методы, определяемые интерфейсом `Optioner`, как это сделано в теле цикла, где вызываются методы `Option.Name()` и `Option.IsValid()`.

Каждый тип параметров имеет поле `Value`, но все они имеют разные типы, например поле `IntOption.Value` имеет тип `int`, а поле `StringOption.Value` – тип `string`. Поэтому для доступа к конкретным полям `Value` (и аналогично к любым другим полям или методам, имеющим определенный тип), необходимо преобразовать текущее значение `option` в значение конкретного типа. Это легко достигается с помощью инструкции `switch выбора по типу` (§5.2.2.2). В выражении

переключателя типа используется *затеняющая переменная* (*option*), которая всегда будет иметь правильный тип для выполняемой инструкции *case* (например, в инструкции *case IntOption* переменная *option* будет иметь тип *IntOption* и т. д.). Поэтому в каждой ветке *case* можно смело обращаться к любым полям и методам с учетом их фактических типов.

6.5. Примеры

Теперь, когда стало понятно, как создавать собственные типы, можно перейти к знакомству с более практичными примерами. Первый пример демонстрирует, как создать простой тип значения. Второй пример демонстрирует, как создать множество взаимосвязанных интерфейсов и структур с применением приема встраивания и как реализовать не только функции-конструкторы, но и *фабричные функции*, чтобы обеспечить возможность создания значений всех типов, экспортируемых пакетом. Третий пример демонстрирует порядок реализации законченного универсального типа коллекций.

6.5.1. Пример: *FuzzyBool* – пользовательский тип с единственным значением

В этом разделе будет показано, как создать пользовательский тип с единственным значением и его методы. Этот пример находится в файле *fuzzy/fuzzybool/fuzzybool.go* и основан на структуре.

Встроенный тип *bool* имеет два возможных значения (*true* и *false*), но в некоторых системах искусственного интеллекта используется нечеткая логика. Логические величины в таких системах могут иметь не только значения «истина» или «ложь», но также значения, занимающие промежуточные положения. В данной реализации для представления логических значений будут использоваться вещественные числа: 0.0 – для значения *false* и 1.0 – для значения *true*. В данной реализации значение 0.5 означает 50% истины (50% лжи), значение 0.25 означает 25% истины (75% лжи) и т. д. Ниже приводятся несколько примеров использования такого типа.

```
func main() {  
    a, _ := fuzzybool.New(0) // Значение ошибки можно смело игнорировать при  
    b, _ := fuzzybool.New(.25) // использовании заведомо допустимых значений, но при
```

```

c, _ := fuzzybool.New(.75) // использовании переменных проверка необходима.
d := c.Copy()
if err := d.Set(1); err != nil {
    fmt.Println(err)
}
process(a, b, c, d)
s := []*fuzzybool.FuzzyBool{a, b, c, d}
fmt.Println(s)
}

func process(a, b, c, d *fuzzybool.FuzzyBool) {
    fmt.Println("Original:", a, b, c, d)
    fmt.Println("Not: ", a.Not(), b.Not(), c.Not(), d.Not())
    fmt.Println("Not Not: ", a.Not().Not(), b.Not().Not(), c.Not().Not(),
        d.Not().Not())
    fmt.Print("0.And(.25)→", a.And(b), " • .25.And(.75)→", b.And(c),
        " • .75.And(1)→", c.And(d), " • .25.And(.75,1)→", b.And(c, d), "\n")
    fmt.Print("0.Or(.25)→", a.Or(b), " • .25.Or(.75)→", b.Or(c),
        " • .75.Or(1)→", c.Or(d), " • .25.Or(.75,1)→", b.Or(c, d), "\n")
    fmt.Println("a < c, a == c, a > c:", a.Less(c), a.Equal(c), c.Less(a))
    fmt.Println("Bool: ", a.Bool(), b.Bool(), c.Bool(), d.Bool())
    fmt.Println("Float: ", a.Float(), b.Float(), c.Float(), d.Float())
}

Original: 0% 25% 75% 100%
Not:      100% 75% 25% 0%
Not Not:  0% 25% 75% 100%
0.And(.25)→0% .25.And(.75)→25% .75.And(1)→75% 0.And(.25,.75,1)→0%
0.Or(.25)→25% .25.Or(.75)→75% .75.Or(1)→100% 0.Or(.25,.75,1)→100%
a < c, a == c, a > c: true false false
Bool:      false false true true
Float:     0 0.25 0.75 1
[0% 25% 75% 100%]

```

Разрабатываемый тип называется `FuzzyBool`. Сначала рассмотрим определение самого типа, затем перейдем к функции-конструктору и закончим изучением методов.

```
type FuzzyBool struct{ value float32 }
```

Тип `FuzzyBool` основан на структуре, содержащей единственное поле типа `float32`. Поле `value` является неэкспортируемым, поэтому для создания значений типа `FuzzyBool` в пакетах, импортирующих пакет `fuzzybool`, необходимо использовать функцию-конструктор (получившую имя `New()` в соответствии с соглашениями, принятыми

в языке Go). Это означает, что мы можем гарантировать создание допустимых значений типа `FuzzyBool`.

Так как тип `FuzzyBool` основан на структуре, содержащей значение с уникальным типом внутри структуры, определение типа можно было бы упростить до `type FuzzyBool struct { float32 }`. При этом пришлось бы изменить выражения доступа к значению (их можно увидеть в реализациях методов, следующих ниже) с `fuzzy.value` на `fuzzy.float32`. Однако предпочтение было отдано именованному полю, потому что такой программный код выглядит более эстетично и отчасти потому, что это потребует внести намного меньше изменений, если позднее потребуется изменить тип поля (например, на `float64`).

Поскольку структура содержит единственное значение, возможны и другие вариации в определении типа. Например, можно было бы определить тип как `type FuzzyBool float32`, основав его непосредственно на типе `float32`. Такой тип тоже с успехом можно было бы использовать, но для работы с ним потребовалось бы писать чуть больше программного кода, и его методы получились бы сложнее, чем в случае подхода на основе структуры, предпринятого здесь. Однако, если сделать значения типа `FuzzyBool` неизменяемыми (то есть для изменения значения использовать не метод `Set()`, а оператор присваивания), определение нового логического типа непосредственно на основе типа `float32` позволило бы существенно упростить программный код.

```
func New(value interface{}) (*FuzzyBool, error) {
    amount, err := float32ForValue(value)
    return &FuzzyBool{amount}, err
}
```

Для удобства пользователей типа `FuzzyBool` инициализировать новые значения этого типа можно не только значениями типа `float32`, но и значениями типа `float64` (тип вещественных чисел, используемый в языке Go по умолчанию), `int` (целочисленный тип по умолчанию) и `bool`. Такая гибкость достигается за счет использования собственной функции `float32ForValue()`, возвращающей значение типа `float32` и `nil` для указанного значения, или `0.0` и значение типа `error`, если функции было передано значение неподдерживаемого типа.

При передаче значения недопустимого типа необходимо немедленно сообщить об ошибке. Но здесь нежелательно генерировать

аварийную ситуацию, которая может вызвать крах приложения. Поэтому вместе со значением типа `*FuzzyBool` возвращается также значение типа `error`. Если функции `New()` передается заведомо допустимый литерал значения (как в примере выше), признак ошибки можно смело игнорировать, но при передаче переменных необходимо обязательно проверять возвращаемое значение типа `error`.

Функция `New()` возвращает указатель на значение типа `FuzzyBool`, а не само значение, потому что было принято решение сделать значения изменяемыми. Это означает, что методам, изменяющим значение типа `FuzzyBool` (в данном примере – единственный метод `Set()`), приемник должен передаваться по *указателю*, а не по значению¹.

Как правило, для неизменяемых типов создаются методы, принимающие приемник по значению, а для изменяемых – по указателю. (Для изменяемых типов вполне возможно реализовать в одних методах передачу приемника по значению, а в других – по указателю, но на практике это не всегда удобно.) Кроме того, указатели лучше подходят для работы с большими составными типами (например, с двумя и более полями), чтобы их значения можно было передавать в виде одного простого указателя.

```
func float32ForValue(value interface{}) (fuzzy float32, err error) {
    switch value := value.(type) { // затеняющая переменная
    case float32:
        fuzzy = value
    case float64:
        fuzzy = float32(value)
    case int:
        fuzzy = float32(value)
    case bool:
        fuzzy = 0
        if value {
            fuzzy = 1
        }
    default:
        return 0, fmt.Errorf("float32ForValue(): %v is not a "+
            "number or Boolean", value)
    }
    if fuzzy < 0 {
        fuzzy = 0
    }
}
```

¹ В действительности можно было бы возвращать значение типа `FuzzyBool` и обеспечить его изменяемость, как показано в примере `fuzzy_value`, который можно найти в загружаемых примерах к книге.

```
} else if fuzzy > 1 {  
    fuzzy = 1  
}  
return fuzzy, nil  
}
```

Эта неэкспортируемая вспомогательная функция используется методами `New()` и `Set()` для преобразования значения `value` в значение типа `float32` в диапазоне `[0.0, 1.0]`. Обработка исходных значений различных типов легко реализуется с помощью инструкции *switch выбора по типу* (§5.2.2.2).

Если функции передается значение недопустимого типа, она возвращает непустое значение типа `error`. Это позволяет вызывающей программе проверить возвращаемое значение и предпринять необходимые действия в случае появления ошибки. Вызывающая программа может возбудить аварийную ситуацию и вызвать крах приложения с выводом трассировочной информации или как-то иначе решить проблему. В низкоуровневых функциях, подобных этой, часто лучше просто возвращать признак ошибки, чтобы сообщить о проблеме, потому что они не обладают достаточной информацией о логике работы приложения и о том, как следует обрабатывать ошибку, тогда как вызывающая программа находится в лучшем положении и знает, какие действия следует предпринять.

Передача значения недопустимого типа определенно является программной ошибкой, и потому имеет смысл возвращать непустое значение типа `error`, но для значений допустимых типов выбран более либеральный подход – значения вне допустимого диапазона просто преобразуются в ближайшее допустимое значение.

```
func (fuzzy *FuzzyBool) String() string {  
    return fmt.Sprintf("%.0f%%", 100*fuzzy.value)  
}
```

Этот метод реализует интерфейс `fmt.Stringer`. Это означает, что значения типа `FuzzyBool` будут выводиться, как указано здесь, и могут передаваться везде, где ожидается значение, реализующее интерфейс `fmt.Stringer`.

Значения типа `FuzzyBool` будут выводиться в виде целого числа процентов. (Напомню, что спецификатор формата `%.0f` обеспечивает вывод вещественных чисел без десятичной точки и без дробной части, а спецификатор `%%` – вывод одного символа `<%>`. О форматировании строк подробно рассказывается в §3.5.)

```
func (fuzzy *FuzzyBool) Set(value interface{}) (err error) {  
    fuzzy.value, err = float32ForValue(value)  
    return err  
}
```

Этот метод обеспечивает изменяемость значений типа `FuzzyBool`. Он очень похож на функцию `New()`, только здесь не создается нового значения, а выполняются операции с существующим значением `*FuzzyBool`. Если полученное значение `value` является недопустимым, вызывающей программе возвращается непустое значение типа `error` и предполагается, что она проверит его.

```
func (fuzzy *FuzzyBool) Copy() *FuzzyBool {  
    return &FuzzyBool{fuzzy.value}  
}
```

Для типов, значения которых передаются по указателю, часто бывает удобно реализовать метод `Copy()`. Данный метод просто создает новое значение типа `FuzzyBool`, копию приемника и возвращает указатель на него. Здесь нет необходимости проверять допустимость исходного значения, так как приемник всегда будет иметь допустимое значение. При этом, разумеется, предполагается, что при создании оригинального значения вызовом функции `New()` и его последующих изменениях с помощью метода `Set()` было получено пустое значение типа `error`.

```
func (fuzzy *FuzzyBool) Not() *FuzzyBool {  
    return &FuzzyBool{1 - fuzzy.value}  
}
```

Это первый метод, реализующий логическую операцию, и, подобно другим методам логических операторов, он принимает приемник типа `*FuzzyBool`.

Реализовать этот метод можно было бы тремя способами. Первый: изменить значение приемника и ничего не возвращать. Второй: изменить значение приемника и вернуть его — этот подход используется многими методами типов `big.Int` и `big.Rat` в стандартной библиотеке. Такой прием позволяет составлять цепочки из операций (например, `b.Not().Not()`). Он также дает возможность экономить память (благодаря повторному использованию существующих значений), но он может превратиться в источник ошибок, если забыть,

что метод возвращает то же значение, относительно которого он вызывался, и что это значение изменяется. Третий способ – тот, что реализован здесь: оставить исходное значение нетронутым и вернуть новое значение типа `FuzzyBool` с результатом логической операции. Этот способ прост, понятен и поддерживает возможность составления цепочек из операций, хотя и ценой создания дополнительных значений. Этот последний способ выбран для всех методов типа `FuzzyBool`, реализующих логические операции.

Логика работы операции НЕ (NOT) проста, она возвращает 1.0 для исходного значения 0.0, 0.0 – для исходного значения 1.0, 0.75 – для исходного значения 0.25, 0.25 – для исходного значения 0.75, 0.5 – для исходного значения 0.5 и т. д.

```
func (fuzzy *FuzzyBool) And(first *FuzzyBool,
    rest ...*FuzzyBool) *FuzzyBool {
    minimum := fuzzy.value
    rest = append(rest, first)
    for _, other := range rest {
        if minimum > other.value {
            minimum = other.value
        }
    }
    return &FuzzyBool{minimum}
}
```

Операция И (AND) возвращает минимальное из указанных значений. Сигнатура метода гарантирует, что его можно будет вызвать как минимум с одним значением (`first`) типа `*FuzzyBool`, при этом ему можно передать нуль или более дополнительных значений (`rest`). Метод просто добавляет значение `first` в конец (возможно, пустого) среза `rest` и затем выполняет итерации по срезу в поисках минимального значения, меньше значения приемника. Подобно методу `Not()` он возвращает новое значение типа `*FuzzyBool`, оставляя нетронутым значение, относительно которого он вызывался.

Операция ИЛИ (OR) возвращает максимальное из указанных значений. Метод `Or()` здесь не показан, потому что структурно он идентичен методу `And()`. Единственное отличие состоит в том, что в методе `Or()` вместо переменной `minimum` используется переменная `maximum` и сравнение выполняется с помощью оператора `<` вместо оператора `>`.

```
func (fuzzy *FuzzyBool) Less(other *FuzzyBool) bool {  
    return fuzzy.value < other.value  
}  
  
func (fuzzy *FuzzyBool) Equal(other *FuzzyBool) bool {  
    return fuzzy.value == other.value  
}  

```

Эти два метода позволяют сравнивать значения типа `FuzzyBool` в терминах вещественных чисел типа `float32`. Оба метода возвращают результат типа `bool`.

```
func (fuzzy *FuzzyBool) Bool() bool {  
    return fuzzy.value >= .5  
}  
  
func (fuzzy *FuzzyBool) Float() float64 {  
    return float64(fuzzy.value)  
}  

```

Функцию-конструктор `fuzzybool.New()` можно рассматривать как функцию преобразования типа, поскольку она может принимать значения типов `float32`, `float64`, `int` и `bool`, и возвращает значение типа `*FuzzyBool`. Два метода выше выполняют обратные преобразования.

Тип `FuzzyBool` представляет собой законченный логический тип данных для применения в системах нечеткой логики, который можно использовать подобно любым другим пользовательским типам. То есть значения типа `*FuzzyBool` можно сохранять в срезах, в виде ключей и значений в отображениях. Разумеется, при использовании значений `*FuzzyBool` в качестве ключей отображений существует возможность сохранить несколько ключей с одинаковыми фактическими значениями, потому что каждое из них будет иметь уникальный адрес в памяти. Одно из решений этой проблемы состоит в том, чтобы использовать фактические значения (как показано в примере `fuzzy_value`, который можно найти в загружаемых примерах к книге). Как вариант можно было бы использовать собственный тип коллекций, хранящих указатели, но выполняющих сравнение по фактическим значениям: подобную возможность предоставляет пользовательский тип `omap.Map`, если передать ему соответствующую функцию сравнения (§6.5.3).

Помимо типа `FuzzyBool`, представленного в этом подразделе, в загружаемых примерах к книге можно также найти три альтернативные реализации: они не показаны и не обсуждаются в книге.

Первые две альтернативы находятся в файлах `fuzzy_value/fuzzybool/fuzzybool.go` и `fuzzy_mutable/fuzzybool/fuzzybool.go` – они обладают точно такой же функциональностью, что и версия, описанная в этом подразделе (находится в файле `fuzzy/fuzzybool/fuzzybool.go`). Версия `fuzzy_value` оперирует значениями типа `FuzzyBool` вместо `*FuzzyBool`, а версия `fuzzy_mutable` основана не на структуре, а непосредственно на типе `float32`. Реализация примера `fuzzy_mutable` получилась немного длиннее и сложнее, чем версия на основе структуры, представленная здесь. Третья версия обладает меньшими функциональными возможностями, потому что она реализует неизменяемый тип `FuzzyBool`. Она также основана непосредственно на типе `float32` и находится в файле `fuzzy_immutable/fuzzybool/fuzzybool.go`. Это самая простая из трех реализаций.

6.5.2. Пример: фигуры – семейство пользовательских типов

Когда имеется множество взаимосвязанных сущностей, таких как геометрические фигуры, к которым можно было бы применить некоторые общие операции (например, нарисовать фигуру), для их реализации можно использовать два основных подхода. Первый, более привычный для программистов на C++, Java и Python, состоит в использовании иерархии типов, или, если говорить в терминах языка Go, встроенных интерфейсов. Однако часто удобнее создать множество независимых интерфейсов, которые затем можно свободно компоновать. В этом подразделе будут показаны оба подхода, первый – в файле `shaper1/shapes/shapes.go` и второй – в файле `shaper2/shapes/shapes.go`. (Обратите внимание, что когда речь будет идти о типах, функциях и методах, одинаковых для обеих реализаций, каковыми являются большинство из них, упоминаться будет просто пакет `shapes`. Естественно, когда необходимо будет подчеркнуть различия, будет упоминаться пакет `shaper1/shapes` или пакет `shaper2/shapes`.)

На рис. 6.3 изображено, что можно получить с помощью пакета `shapes`, – в данном случае с помощью пакета был нарисован белый прямоугольник внутри круга и несколько многоугольников с различным количеством сторон и разных цветов.



Рис. 6.3. Пример использования пакета `shaper`; файл `shapes.png`

Пакет `shapes` экспортирует три функции для работы с изображениями и определяет три типа фигур, из которых экспортируются только два. Пакет `shapes1/shapes`, иллюстрирующий иерархическое решение, экспортирует три интерфейса, а пакет `shapes2/shapes`, иллюстрирующий композиционное решение, экспортирует пять интерфейсов. Для начала рассмотрим вспомогательные функции, предназначенные для работы с изображениями, затем перейдем к интерфейсам (они будут рассматриваться в двух отдельных подразделах) и, наконец, рассмотрим программный код для работы с фигурами.

6.5.2.1. Вспомогательные функции

Пакет `image` из стандартной библиотеки экспортирует интерфейс `image.Image`. Этот интерфейс определяет три метода: `image.Image.ColorModel()`, возвращающий цветовую модель изображения (как значение типа `color.Model`), `image.Image.Bounds()`, возвращающий прямоугольник, ограничивающий область изображения (как значение типа `image.Rectangle`), и `image.Image.At(x, y)`, возвращающий значение цвета типа `color.Color` для указанного пикселя. Примечательно, что интерфейс `image.Image` не определяет метода для изменения пикселя, даже при том, что некоторые типы в пакете `image` имеют метод `Set(x, y int, fill color.Color)`. Однако в пакете `image/draw` имеется интерфейс `draw.Image`, встраивающий интерфейс `image.Image`, который определяет метод `Set()`. Среди прочих интерфейс `draw.Image` реализуется типами `image.Gray` и `image.RGBA` из стандартной библиотеки.

```
func FilledImage(width, height int, fill color.Color) draw.Image {
    if fill == nil { // Значение nil цвета просто интерпретируется как черный цвет
        fill = color.Black
    }
    width = saneLength(width)
    height = saneLength(height)
    img := image.NewRGBA(image.Rect(0, 0, width, height))
    draw.Draw(img, img.Bounds(), &image.Uniform{fill}, image.ZP, draw.Src)
    return img
}
```

Это экспортируемая вспомогательная функция, которая создает изображение указанного размера, равномерно залитое указанным цветом.

Сначала функция замещает значение `nil` цвета черным цветом и подгоняет оба размера под допустимые значения. Затем она создает значение типа `image.RGBA` (изображение, цвета в котором определяются в виде значений красной, зеленой и синей составляющих, а также степени прозрачности) и возвращает его как значение типа `draw.Image`, поскольку нам важны функциональные возможности, а не фактический тип.

Функция `draw.Draw()` принимает целевое изображение (типа `draw.Image`), прямоугольник области рисования (в данном случае все изображение), исходное изображение для копирования (в данном случае изображение с бесконечными размерами, залитое указанным цветом), координаты прямоугольника для рисования (`image.ZP` – это нулевая точка, то есть точка с координатами `(0, 0)`) и метод рисования. Здесь выбран метод `draw.Src`, поэтому функция просто скопирует исходное изображение в целевое. То есть функция окрасит каждый пиксель целевого исходного изображения указанным цветом. (В пакете `draw` имеется также функция `draw.DrawMask()`, поддерживающая возможность применения различных правил слияния пикселей.)

```
var saneLength, saneRadius, saneSides func(int) int
func init() {
    saneLength = makeBoundedIntFunc(1, 4096)
    saneRadius = makeBoundedIntFunc(1, 1024)
    saneSides = makeBoundedIntFunc(3, 60)
}
```

Здесь определяются три неэкспортируемые переменные для хранения ссылок на вспомогательные функции, каждая из которых принимает значение типа `int` и возвращает значение типа `int`. Кроме того, для данного пакета реализована функция `init()`, где переменным присваиваются ссылки на соответствующие анонимные функции.

```
func makeBoundedIntFunc(minimum, maximum int) func(int) int {
    return func(x int) int {
        valid := x
        switch {
        case x < minimum:
            valid = minimum
        case x > maximum:
            valid = maximum
        }
    }
}
```

```

    if valid != x {
        log.Printf("%s(): replaced %d with %d\n", caller(1), x, valid)
    }
    return valid
}
}

```

Эта функция возвращает другую функцию, возвращающую указанное значение *x*, если оно находится между значениями *minimum* и *maximum* (включительно), или ближайшее граничное значение.

Если значение *x* находится за границами диапазона, функция не только возвращает допустимое альтернативное значение, но еще и регистрирует проблему в журнале. Однако в сообщении об обнаруженной проблеме не должно фигурировать имя функции, созданной здесь (то есть *saneLength()*, *saneRadius()* или *saneSides()*), потому что фактически проблема рождается в вызывающих их функциях. Поэтому вместо имени функции, созданной здесь, в журнал выводится имя вызывающей функции, которое возвращает функция *caller()*.

```

func caller(steps int) string {
    name := "?"
    if pc, _, _, ok := runtime.Caller(steps + 1); ok {
        name = filepath.Base(runtime.FuncForPC(pc).Name())
    }
    return name
}

```

Функция *runtime.Caller()* возвращает информацию о функции, которая была вызвана в текущей *go*-подпрограмме, но еще не вернула управление. Ее аргумент типа *int* сообщает, на сколько шагов (то есть функций) назад следует заглянуть. При значении 0 аргумента возвращается информация о текущей функции (то есть о самой функции *shapes.caller()*), при значении 1 возвращается информация о вызвавшей ее функции и т. д. Здесь к значению аргумента добавляется 1, чтобы сразу начать с вызывающей функции.

Функция *runtime.Caller()* возвращает четыре значения: программный счетчик (сохраняется в переменной *pc*), имя файла и номер строки, где произошел вызов (оба значения игнорируются за счет присваивания *пустым идентификаторам*), и логический флаг (сохраняется в переменной *ok*), сообщающий об успешной или неудачной попытке извлечения информации.

В случае успеха программный счетчик передается функции `runtime.FuncForPC()`, возвращающей значение типа `*runtime.Func`, которое затем передается методу `runtime.Func.Name()` для получения имени вызывающей функции. Возвращаемое имя имеет вид пути к файлу, например: `/home/mark/goeg/src/shaper1/shapes.FilledRectangle` — для функции, или `/home/mark/goeg/src/shaper1/shapes.*shape.SetFill` — для метода. Для небольших проектов путь к файлу не представляет интереса, поэтому здесь он отбрасывается с помощью функции `filepath.Base()`. После этого имя возвращается вызывающей программе.

Например, если вызвать функцию `shapes.FilledImage()` и передать ей значение, выходящее за допустимый диапазон, такое как 5000, проблема будет исправлена в функции `saneLength()`. Кроме того, проблема будет зафиксирована в журнале в виде сообщения: `"shapes.FilledRectangle(): replaced 5000 with 4096"`. Это обусловлено тем, что `saneLength()` вызовет `caller()` со значением аргумента 1, к которому функция `caller()` прибавит 1 и запросит информацию о третьей функции, вверх по стеку: 0 — сама функция `caller()`, 1 — `saneLength()` и 2 — `FilledImage()`.

```
func DrawShapes(img draw.Image, x, y int, shapes ...Shaper) error {
    for _, shape := range shapes {
        if err := shape.Draw(img, x, y); err != nil {
            return err
        }
    }
    return nil
}
```

Это еще одна экспортируемая вспомогательная функция, и единственная, реализованная в разных пакетах `shapes` по-разному. Выше представлена версия из пакета `shapes1/shapes`, реализующего иерархический подход. Функция в пакете `shapes2/shapes`, реализующем композиционный подход, отличается только сигнатурой, где она принимает значения типа `Drawer`, то есть реализующие интерфейс `Drawer` (имеющие метод `Draw()`), вместо значений типа `Shaper`, которые должны иметь методы `Draw()`, `Fill()` и `SetFill()`. Другими словами, при композиционном подходе используется более специализированный и менее требовательный тип аргумента (`Drawer`), чем при иерархическом (требующем реализации интерфейса `Shaper`). Эти интерфейсы будут рассматриваться в следующих двух подподразделах.

Тело функции и ее поведение в обоих случаях ничем не отличаются. Функция принимает изображение `draw.Image`, в котором выполняется рисование, позицию (координаты x и y) и нуль или более значений типа `Shaper` (или `Drawer`). Внутри цикла каждой фигуре предлагается нарисовать себя в изображении, в указанной позиции. В процессе рисования координаты x и y проверяются в низкоуровневом методе `Draw()` фигуры, и если они оказываются недопустимыми, метод возвращает непустой признак ошибки, который немедленно передается вызывающей программе.

При создании изображения на рис. 6.3 (выше) использовалась модифицированная версия этой функции, которая рисует каждую фигуру три раза. Первый раз – в указанных координатах x и y , второй – со смещением на один пиксель вправо и третий – со смещением на один пиксель вниз. Это было сделано, чтобы увеличить толщину линий на снимке с экрана.

```
func SaveImage(img image.Image, filename string) error {
    file, err := os.Create(filename)
    if err != nil {
        return err
    }
    defer file.Close()
    switch strings.ToLower(filepath.Ext(filename)) {
    case ".jpg", ".jpeg":
        return jpeg.Encode(file, img, nil)
    case ".png":
        return png.Encode(file, img)
    }
    return fmt.Errorf("shapes.SaveImage(): '%s' has an unrecognized "+
        "suffix", filename)
}
```

Это последняя из экспортируемых вспомогательных функций. Она получает значение изображения, реализующее интерфейс `image.Image`, которым может быть любое значение, реализующее интерфейс `draw.Image`, так как он встраивает интерфейс `image.Image`. Функция пытается сохранить изображение в файл с указанным именем. Если функция `os.Create()` потерпит неудачу (например, из-за того, что было указано пустое имя файла, или из-за ошибки ввода/вывода), или имя файла имеет неподдерживаемое расширение, или потерпела неудачу попытка кодирования изображения, функция возвращает непустой признак ошибки.

На момент написания этих строк стандартная библиотека языка Go поддерживала для чтения и записи два формата файлов с изображениями: `.png` (Portable Network Graphics – переносимая сетевая графика) и `.jpg` (Joint Photographic Experts Group – объединенная экспертная группа по фотографии). Пакеты поддержки дополнительных форматов изображений доступны на сайте godashboard.appspot.com/project. Функция `jpeg.Encode()` имеет дополнительный аргумент, который можно использовать для тонкой настройки параметров сохранения изображения – здесь в этом аргументе передается `nil`, что соответствует настройкам по умолчанию.

Эти функции кодирования могут возбуждать аварийную ситуацию, например если вместо значения типа `image.Image` передать `nil`. Поэтому, чтобы защитить программу от ошибок, можно было бы предусмотреть отложенный вызов функции, вызывающей функцию `recover()`, либо в этой функции, либо в одной из вызывающих функций (§5.5.1). Здесь было решено *не* добавлять такую защиту, потому что контрольные тесты (здесь не показаны) производят достаточное количество вызовов функции, и можно с уверенностью сказать, что ошибка проявится при первом же тестировании и вызовет аварийное завершение программы. Поэтому ее практически невозможно будет пропустить.

Благодаря поддержке интерфейса `draw.Image` можно определять значения изображений, позволяющие изменять цвет отдельных пикселей. С помощью функции `DrawShapes()` можно рисовать фигуры (значения, реализующие интерфейс `Shaper` или `Drawer`) в таких изображениях. А вызовом функции `SaveImage()` можно сохранять изображения на диске. Помимо этих вспомогательных функций, нам необходимо определить интерфейсы (такие как `Shaper`, `Drawer` и др.), а также конкретные типы с их методами реализации интерфейсов.

6.5.2.2. Иерархия встроенных интерфейсов

Программисты с опытом обычного объектно-ориентированного программирования на основе наследования наверняка предпочтут использовать встроенные интерфейсы для создания их иерархии. Рекомендуемый же способ основан на композиционной модели, которая будет рассматриваться в следующем подразделе, а здесь мы познакомимся с иерархической моделью, реализованной в пакете `shapes1/shapes`.

```
type Shaper interface {
    Fill() color.Color
    SetFill(fill color.Color)
    Draw(img draw.Image, x, y int) error
}

type CircularShaper interface {
    Shaper // Fill(); SetFill(); Draw()
    Radius() int
    SetRadius(radius int)
}

type RegularPolygonShaper interface {
    CircularShaper // Fill(); SetFill(); Draw(); Radius(); SetRadius()
    Sides() int
    SetSides(sides int)
}
```

Здесь путем встраивания, не наследования, создается иерархия из трех интерфейсов, определяющих методы для фигур.

Интерфейс `Shaper` определяет методы для получения и изменения цвета заливки — значений типа `color.Color`, а также метод рисования фигуры в заданной позиции в изображении `draw.Image`. Интерфейс `CircularShaper` встраивает интерфейс `Shaper`, а также добавляет методы получения и изменения значения радиуса типа `int`. Аналогично интерфейс `RegularPolygonShaper` встраивает интерфейс `CircularShaper` (и через него интерфейс `Shaper`) и добавляет методы получения и изменения числа сторон типа `int`.

Несмотря на то что подход на основе создания иерархий, подобных этой, может быть многим знаком и дает положительные результаты, тем не менее это не самый лучший способ решения задач на языке Go. Причина в том, что такой подход запирает программиста в рамках иерархии, даже когда в иерархии нет никакой необходимости: в действительности здесь необходимо лишь обеспечить реализацию подходящих интерфейсов в фигурах. Это даст больше гибкости, как будет показано в следующем подразделе.

6.5.2.3. Свободно компокуемые независимые интерфейсы

В фигурах основной интерес представляют операции, которые можно выполнять над ними (рисование, получение/изменение цвета заливки, получение/изменение величины радиуса и т. д.), обладающие некоторой долей универсальности. Ниже представлены интерфейсы,

объявленные в пакете `shapes2/shapes`, реализующем композиционный подход.

```
type Shaper interface {
    Drawer // Draw()
    Filler // Fill(); SetFill()
}
type Drawer interface {
    Draw(img draw.Image, x, y int) error
}
type Filler interface {
    Fill() color.Color
    SetFill(fill color.Color)
}
type Radiuser interface {
    Radius() int
    SetRadius(radius int)
}
type Sideser interface {
    Sides() int
    SetSides(sides int)
}
```

Интерфейс `Shaper` в этом пакете демонстрирует простой способ определения обобщенной фигуры, то есть фигуры, которую можно нарисовать и которая позволяет получить/изменить цвет заливки. Все остальные интерфейсы определяют более специализированные особенности (получение и изменение дополнительных параметров фигур).

Множество независимых интерфейсов обеспечивают большую гибкость, чем их иерархия. Например, при использовании функции `DrawShapes()` можно более точно определять передаваемые ей фигуры, чем это было возможно в иерархической модели. Кроме того, в отсутствие иерархии можно более свободно добавлять другие интерфейсы и, конечно же, при наличии множества узкоспециализированных интерфейсов их проще компоновать для достижения требуемого результата, как это видно на примере интерфейса `Shaper`.

Две разные версии пакета `shapes` имеют совершенно отличные интерфейсы (даже при том, что в обоих имеется интерфейс `Shaper`, их тела совершенно отличаются). Тем не менее, поскольку интерфейсы и конкретные типы полностью независимы друг от друга, эти различия практически не оказывают влияния на конкретную реализацию интерфейсов.

6.5.2.4. Конкретные типы и методы

Это последний подподраздел с описанием пакета `shapes` – здесь будут представлены конкретные реализации интерфейсов, описанных в двух предыдущих подподразделах.

```
type shape struct{ fill color.Color }  
func newShape(fill color.Color) shape {  
    if fill == nil { // Значение nil цвета просто интерпретируется как черный цвет  
        fill = color.Black  
    }  
    return shape{fill}  
}  
func (shape shape) Fill() color.Color { return shape.fill }  
func (shape *shape) SetFill(fill color.Color) {  
    if fill == nil { // Значение nil цвета просто интерпретируется как черный цвет  
        fill = color.Black  
    }  
    shape.fill = fill  
}
```

Этот простой тип является неэкспортируемым, поэтому он доступен только внутри пакета `shapes`. То есть значение типа `shape` не может быть создано за пределами пакета.

В иерархическом пакете `shaper1/shapes` этот тип не реализует ни один из интерфейсов из-за отсутствия метода `Draw()`. Но в композиционном пакете `shaper2/shapes` он реализует интерфейс `Filler`.

В программном коде только тип `Circle` (который рассматривается чуть ниже) встраивает интерфейс `shape` непосредственно. Поэтому теоретически в тип `Circle` можно было бы добавить поле типа `color.Color`, реализовать методы получения и изменения цвета, принимающие приемник типа `*Circle` вместо типа `*shape` и полностью избавиться от типа `shape`. Однако для большей гибкости лучше оставить тип `shape`, потому что это упрощает добавление дополнительных интерфейсов и типов фигур, которые могут основываться непосредственно на типе `shape` (то есть иметь цвет), а не на типе `Circle` (например, из-за неприменимости к ним понятия радиус). Эта гибкость еще пригодится при выполнении одного из упражнений.

```
type Circle struct {  
    shape  
    radius int  
}
```

```

}
func NewCircle(fill color.Color, radius int) *Circle {
    return &Circle{newShape(fill), saneRadius(radius)}
}
func (circle *Circle) Radius() int {
    return circle.radius
}
func (circle *Circle) SetRadius(radius int) {
    circle.radius = saneRadius(radius)
}
func (circle *Circle) Draw(img draw.Image, x, y int) error {
    // ... около 30 строк ...
}
func (circle *Circle) String() string {
    return fmt.Sprintf("circle(fill=%v, radius=%d)", circle.fill,
        circle.radius)
}

```

Это полная реализация типа `Circle`. Несмотря на возможность создавать значения конкретного типа `*Circle`, их можно передавать как интерфейсы, что обеспечивает значительную гибкость. Например, функция `DrawShapes()` принимает значения, реализующие интерфейс `Shaper` (или `Drawer`), независимо от их конкретного типа.

В иерархическом пакете `shaper1/shapes` этот тип реализует интерфейсы `CircularShaper` и `Shaper`. В композиционном пакете `shaper2/shapes` он реализует интерфейсы `Filler`, `Radiuser`, `Drawer` и `Shaper`. И в обоих пакетах этот тип также реализует интерфейс `fmt.Stringer`.

Поскольку в языке Go отсутствуют конструкторы типов и в определении типа имеются неэкспортируемые поля, необходимо реализовать функцию-конструктор, которая должна вызываться явно. Функция-конструктор для типа `Circle` называется `NewCircle()` – далее можно будет увидеть, что в пакете имеется также функция `New()`, способная создавать значения любых типов фигур из пакета `shapes`. Вспомогательная функция `saneRadius()`, созданная выше, возвращает переданное ей целое число, если оно попадает в заданный диапазон, в противном случае – ближайшее граничное значение диапазона.

Реализация метода `Draw()` была опущена (ее можно найти в загружаемых примерах к книге), так как основной интерес в этой главе представляют интерфейсы и типы, а не приемы работы с графикой.


```
type RegularPolygon struct {  
    *Circle  
    sides int  
}  
func NewRegularPolygon(fill color.Color, radius,  
    sides int) *RegularPolygon {  
    return &RegularPolygon{NewCircle(fill, radius), saneSides(sides)}  
}  
func (polygon *RegularPolygon) Sides() int {  
    return polygon.sides  
}  
func (polygon *RegularPolygon) SetSides(sides int) {  
    polygon.sides = saneSides(sides)  
}  
func (polygon *RegularPolygon) Draw(img draw.Image, x, y int) error {  
    // ... примерно 55 строк, включая две вспомогательные функции ...  
}  
func (polygon *RegularPolygon) String() string {  
    return fmt.Sprintf("polygon(fill=%v, radius=%d, sides=%d)",  
        polygon.Fill(), polygon.Radius(), polygon.sides)  
}
```

Это полная реализация типа `RegularPolygon`, представляющего обычные многоугольники. Этот тип очень похож на тип `Circle`, только имеет более сложную реализацию метода `Draw()` (тело которого было опущено). Поскольку тип `RegularPolygon` встраивает тип `*Circle`, часть его полей заполняется с помощью функции `NewCircle()` (выполняющей необходимые проверки). Вспомогательная функция `saneSides()` действует точно так же, как функции `saneRadius()` и `saneLength()`.

В иерархическом пакете `shaper1/shapes` этот тип реализует интерфейсы `RegularPolygonalShaper`, `CircularShaper`, `Shaper` и `fmt.Stringer`. В композиционном пакете `shaper2/shapes` он реализует интерфейсы `Filler`, `Radiuser`, `Sideser`, `Drawer`, `Shaper` и `fmt.Stringer`.

Функции `NewCircle()` и `NewRegularPolygon()` позволяют создавать значения типов `*Circle` и `*RegularPolygon`, а поскольку эти типы реализуют интерфейс `Shaper` и другие, их значения можно передавать как реализующие интерфейс `Shapers` или другие интерфейсы. Относительно таких значений можно вызывать любые методы интерфейса `Shaper` (то есть `Fill()`, `SetFill()` и `Draw()`). А если для значения типа `Shaper` потребуется вызвать метод, не принадлежащий интерфейсу `Shaper`, с помощью операции приведения типа или выбора по типу можно будет получить доступ к значению как к значению,

реализующему требуемый интерфейс. Пример такого подхода будет показан ниже, при знакомстве с функциями `showShapeDetails()`.

Легко можно представить, что впоследствии может потребоваться определить другие типы фигур, из которых одни будут основаны на типе `shape`, другие — на типах `Circle` и `RegularPolygon`. Кроме того, могут возникнуть ситуации, когда потребуется создавать фигуры, форма которых определяется во время выполнения, например в виде названия фигуры. Для этой цели можно создать *фабричную функцию*, то есть функцию, возвращающую значения фигур, конкретный тип которых зависит от аргумента.

```

type Option struct {
    Fill    color.Color
    Radius int
}

func New(shape string, option Option) (Shaper, error) {
    sidesForShape := map[string]int{"triangle": 3, "square": 4,
        "pentagon": 5, "hexagon": 6, "heptagon": 7, "octagon": 8,
        "enneagon": 9, "nonagon": 9, "decagon": 10}
    if sides, found := sidesForShape[shape]; found {
        return NewRegularPolygon(option.Fill, option.Radius, sides), nil
    }
    if shape != "circle" {
        return nil, fmt.Errorf("shapes.New(): invalid shape '%s'", shape)
    }
    return NewCircle(option.Fill, option.Radius), nil
}

```

Эта фабричная функция принимает два аргумента: название фигуры, которую требуется создать, и значение типа `Option` с дополнительными параметрами фигуры. (Об использовании структур для поддержки необязательных аргументов рассказывалось в главе 5, в §5.6.1.3.) Функция возвращает фигуру, реализующую интерфейс `Shaper` и значение `nil` или, если указано недопустимое название фигуры, `nil` и значение ошибки. (Напомню, что интерфейс `Shaper` в разных пакетах `shapes` объявлен по-разному, §6.5.2.2 и §6.5.2.3.) Выбор того или иного типа фигуры производится на основе аргумента `shape` с названием фигуры. Здесь нет необходимости проверять цвет или радиус, потому что эти проверки будут выполнены методом `shapes.shape.SetFill()` и функцией `shapes.saneRadius()`, которые в конечном итоге будут вызваны методами `NewRegularPolygon()` и `NewCircle()`. То же касается и количества сторон.

```

polygon := shapes.NewRegularPolygon(color.RGBA{0, 0x7F, 0, 0xFF}, 65, 4)
showShapeDetails(polygon) ❶
y = 30
for i, radius := range []int{60, 55, 50, 45, 40} {
    polygon.SetRadius(radius)
    polygon.SetSides(i + 5)
    x += radius
    y += height / 8
    if err := shapes.DrawShapes(img, x, y, polygon); err != nil {
        fmt.Println(err)
    }
}
}

```

Этот короткий фрагмент демонстрирует, как с помощью функции `DrawShapes()` создавались некоторые многоугольники, изображенные на рис. 6.3 (выше). Функция `showShapeDetails()` (❶ в листинге выше) используется для вывода информации о фигурах *любых* типов. Это возможно благодаря тому, что функция принимает любые значения, реализующие интерфейс `Shaper` (то есть любые фигуры), а не значения конкретных типов фигур (таких как `*Circle` или `*RegularPolygon`).

Поскольку в двух пакетах `shapes` интерфейс `Shaper` объявлен по-разному, имеются две разные реализации функции `showShapeDetails()`. Ниже показана реализация из иерархической версии пакета `shaper1/shapes`.

Встраивание – это не наследование

В этом подразделе на примере пакета `shaper` демонстрируется, как можно использовать прием *встраивания* для достижения эффекта, напоминающего *наследование*. Этот прием может показаться привлекательным для тех, кто занимается переносом программ с языка C++ или Java на язык Go (или программистам с опытом работы на C++ или Java, изучающим язык Go). Однако хотя этот прием дает положительные результаты, тем не менее *в программах на языке Go следует не имитировать наследование, а вообще избегать его*.

В контексте примера это означает необходимость создания независимых структур:

```

type Circle struct {      type RegularPolygon struct {
    color.Color            color.Color
    Radius int             Radius int
                           Sides int
}
                           }

```

Такой подход все еще позволяет передавать обобщенные значения, представляющие фигуры. В конце концов, если обе фигуры имеют

методы `Draw()`, реализующие интерфейс `Drawer`, значит, значения обоих типов, `Circle` и `RegularPolygon`, можно передавать как значения типа `Drawer`.

Важно также отметить, что все поля здесь объявлены экспортируемыми и без поддержки проверки их значений. Это означает, что их значения должны проверяться в момент использования, а не в момент изменения. В принципе, оба подхода к контролю допустимости значений хороши: выбор того или иного зависит от конкретных требований.

Представленные выше структуры используются в примере `shaper3`, который обладает той же функциональностью, что и примеры `shaper1` и `shaper2`, обсуждаемые в этом разделе. Однако пакет `shaper3` написан в более характерном для языка Go стиле, без встраивания и с реализацией проверки значений полей в момент их использования.

```
func showShapeDetails(shape shapes.Shaper) {
    fmt.Println("fill=", shape.Fill(), " ") // Все фигуры имеют цвет заливки
    if shape, ok := shape.(shapes.CircularShaper); ok { // затеняющая переменная
        fmt.Println("radius=", shape.Radius(), " ")
        if shape, ok := shape.(shapes.RegularPolygonalShaper); ok { // затеняющая
            fmt.Println("sides=", shape.Sides(), " ")
        }
    }
    fmt.Println()
}
```

В иерархии интерфейсов, в пакете `shaper1/shapes`, методы `Fill()` и `SetFill()` определяются интерфейсом `Shaper`, поэтому они могут вызываться непосредственно. Но для вызова других методов используется контролируемое приведение типа, чтобы убедиться, что переданное значение `shape` реализует необходимые интерфейсы. Здесь, например, метод `Radius()` вызывается, только если значение `shape` реализует интерфейс `CircularShaper`. То же касается и метода `Sides()`, объявляемого интерфейсом `RegularPolygonalShaper`. (Напомню, что интерфейс `RegularPolygonalShaper` встраивает интерфейс `CircularShaper`.)

Версия функции `showShapeDetails()` из пакета `shaper2/shapes` похожа на версию из пакета `shaper1/shapes`.

```
func showShapeDetails(shape shapes.Shaper) {
    fmt.Println("fill=", shape.Fill(), " ") // Все фигуры имеют цвет заливки
    if shape, ok := shape.(shapes.Radiuser); ok { // затеняющая переменная
        fmt.Println("radius=", shape.Radius(), " ")
    }
```

```
}  
if shape, ok := shape.(shapes.Sideser); ok { // затеняющая переменная  
    fmt.Println("sides=", shape.Sides(), " ")  
}  
fmt.Println()  
}
```

Композиционный пакет `shaper2/shapes` определяет вспомогательный интерфейс `Shaper`, встраивающий интерфейсы `Drawer` и `Filler`, поэтому заранее известно, что переданное значение `shape` имеет метод `Fill()`. И в отличие от версии в пакете `shaper1/shapes`, здесь можно использовать более специализированные операции приведения типа для доступа к методам `Radius()` и `Sides()` фигур, реализующих их.

Если в тип `shape`, `Circle` или `RegularPolygon` добавить новые поля или методы, это не потребует вносить изменения в существующий программный код. Но если добавить новые методы в любой из интерфейсов, придется реализовать их в соответствующих типах фигур, иначе программный код не будет работать. Более удачное решение состоит в том, чтобы объявить новые интерфейсы с дополнительными методами и встроить в них существующие интерфейсы. Это не нарушит работу имеющегося программного кода и даст возможность выбирать – добавлять или не добавлять реализацию новых методов в существующие типы, в зависимости от необходимости реализации новых интерфейсов вдобавок к уже реализованным.

При работе с *интерфейсами* рекомендуется применять композиционный подход, а не иерархический. Что касается структур, рекомендуется использовать прием, характерный для языка Go, то есть создавать независимые структуры и не пытаться имитировать наследование. Разумеется, после обретения достаточного опыта такой выбор будет производиться, исходя из культуры программирования на языке Go, а не на основе простоты переноса с других языков или из-за привычки.

В дополнение к примерам `shaper1` и `shaper2`, представленным в этом подразделе, в загружаемых примерах к книге имеется также пример `shaper3`, демонстрирующий «чистокровный» способ реализации на языке Go. Версия `shaper3` имеет всего один интерфейс, `Drawer`, и независимые структуры `Circle` и `RegularPolygon` (как показано во врезке «Встраивание – это не наследование» выше). Кроме того, в версии `shaper3` используются фактические значения типа `shape`, а не указатели, а проверка значений полей выполняется в момент их использования. Обязательно загляните в файлы `shaper2/shapes/shapes.go` и `shaper3/shapes/shapes.go` и сравните эти два подхода.

6.5.3. Пример: упорядоченное отображение – обобщенный тип коллекций

В последнем примере для этой главы представлен обобщенный тип упорядоченных отображений, который хранит пары *ключ/значение* подобно встроенному типу `map`, но все пары хранятся в порядке следования ключей. В реализации упорядоченного отображения используется левостороннее красно-черное дерево, благодаря чему достигается высокая скорость работы, где алгоритм поиска имеет сложность $O(\log_2 n)$ ¹. Для сравнения, производительность алгоритма на основе несбалансированных двоичных деревьев может ухудшаться до производительности алгоритма работы со связанными списками ($O(n)$), если элементы добавляются по порядку. Сбалансированные деревья не страдают от этого дефекта, потому что они поддерживают сбалансированность при добавлении и удалении элементов, благодаря чему сохраняется высокая производительность.

Программисты с опытом объектно-ориентированного программирования на основе наследования (на таких языках, как C++, Java, Python) наверняка решат заставить ключи упорядоченного отображения поддерживать оператор сравнения `<` или метод `Less(другой_ключ)` `bool`. Этого легко можно добиться, определив интерфейс `Lesser`, требующий реализации такого метода, и объявив типы-обертки для типов `int`, `string`, `MyType` и т. д., реализующие этот метод. Однако в языке Go используется несколько иной подход.

В данной реализации упорядоченных отображений не накладываются ограничений на тип ключей. Вместо этого каждое отображение будет снабжаться функцией «меньше чем» для сравнения ключей. То есть совершенно не важно, будут ли поддерживать ключи оператор `<` или нет, — главное, чтобы имела подходящая функция, способная сравнивать их.

¹ Представленная здесь реализация упорядоченных отображений основана на левосторонних красно-черных деревьях, описанных Робертом Седжвиком (Robert Sedgwick) в его статьях www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf и www.cs.princeton.edu/~rs/talks/LLRB/RedBlack.pdf. На момент написания этих строк реализации алгоритмов на языке Java, представленные в этих статьях, были неполными и содержали ошибки, поэтому здесь были использованы идеи Ли Станца (Lee Stanza) из его реализации алгоритма на C++, которую можно найти на сайте www.teach-solaisgames.com/articles/balanced_left_leaning.html.

Прежде чем приступить к обсуждению реализации, рассмотрим несколько примеров ее использования, начав с создания и заполнения упорядоченного отображения.

```
words := []string{"Puttering", "About", "in", "a", "Small", "Land"}
wordForWord := omap.NewCaseFoldedKeyed()
for _, word := range words {
    wordForWord.Insert(word, strings.ToUpper(word))
}

```

Реализация упорядоченного отображения находится в пакете `omap` в виде типа `Map`. Чтобы создать значение типа `Map` необходимо вызвать функцию `omap.New()` или любую другую функцию-конструктор, возвращающую значение типа `Map`, такую как `omap.NewCaseFoldedKeyed()`, использованную здесь, потому что нулевое значение типа `Map` непригодно к использованию. Данная конкретная функция-конструктор создает пустое отображение типа `Map` с предопределенной функцией сравнения строковых ключей без учета регистра символов и возвращает указатель (то есть значение типа `*Map`).

Каждая пара *ключ/значение* добавляется с помощью метода `omap.Map.Insert()`, принимающего два значения типа `interface{}`, то есть ключ и значение любых типов. (Однако тип ключа должен быть совместим с функцией сравнения, поэтому в данном примере используются строковые ключи.) Метод `Insert()` возвращает `true`, если новый элемент был успешно вставлен, и `false` – если элемент с указанным ключом уже существует (в этом случае существующее значение элемента замещается новым значением – точно так же действует и встроенный тип `map`).

```
wordForWord.Do(func(key, value interface{}) {
    fmt.Printf("%v→%v\n", key, value)
})
a→A
About→ABOUT
in→IN
Land→LAND
Puttering→PUTTERING
Small→SMALL

```

Метод `omap.Map.Do()` принимает функцию с сигнатурой `func(interface{}, interface{})` и вызывает ее для каждого элемента

в упорядоченном отображении, в порядке следования ключей, передавая ключи и значения в виде аргументов. Здесь метод `Do()` был использован для вывода всех ключей и значений в отображении `wordForWord`.

Помимо добавления элементов и вызова функции для каждого элемента в отображении, имеется также возможность получать количество элементов, отыскивать требуемые элементы и удалять элементы.

```
fmt.Println("length before deleting:", wordForWord.Len())
_, containsSmall := wordForWord.Find("small")
fmt.Println("contains small:", containsSmall)
for _, key := range []string{"big", "medium", "small"} {
    fmt.Printf("%t ", wordForWord.Delete(key))
}
_, containsSmall = wordForWord.Find("small")
fmt.Println("\nlength after deleting: ", wordForWord.Len())
fmt.Println("contains small:", containsSmall)
length before deleting: 6
contains small: true
false false true
length after deleting: 5
contains small: false
```

Метод `omap.Map.Len()` возвращает количество элементов в упорядоченном отображении.

Метод `omap.Map.Find()` возвращает значение элемента с указанным ключом в виде значения типа `interface{}` и `true`, или `nil` и `false`, если искомый элемент отсутствует. Метод `omap.Map.Delete()` удаляет элемент с указанным ключом и возвращает `true`. Если требуемый элемент отсутствует в отображении, метод ничего не делает и возвращает `false`.

Если потребуется использовать ключи пользовательского типа, это можно сделать, создав отображение типа `Map` с помощью функции `omap.New()` и реализовав соответствующий метод сравнения.

Например, ниже приводится реализация очень простого пользовательского типа.

```
type Point struct{ X, Y int }
func (point Point) String() string {
    return fmt.Sprintf("(%d, %d)", point.X, point.Y)
}
```

Теперь посмотрим, как создать упорядоченное отображение с ключами типа `*Point` и значениями, определяющими расстояние от начала координат.

В следующем фрагменте создается пустое отображение типа `Map`, которому передается функция сравнения ключей `*Point`. Затем создается срез со значениями `*Point` и выполняется заполнение отображения. И наконец, с помощью метода `omap.Map.Do()` выводятся ключи и значения, хранящиеся в отображении, в порядке следования ключей.

```
distanceForPoint := omap.New(func(a, b interface{}) bool {
    α, β := a.(*Point), b.(*Point)
    if α.X != β.X {
        return α.X < β.X
    }
    return α.Y < β.Y
})
points := []*Point{{3, 1}, {1, 2}, {2, 3}, {1, 3}, {3, 2}, {2, 1}, {2, 2}}
for _, point := range points {
    distance := math.Hypot(float64(point.X), float64(point.Y))
    distanceForPoint.Insert(point, distance)
}
distanceForPoint.Do(func(key, value interface{}) {
    fmt.Printf("%v → %.2v\n", key, value)
})
(1, 2) → 2.2
(1, 3) → 3.2
(2, 1) → 2.2
(2, 2) → 2.8
(2, 3) → 3.6
(3, 1) → 3.2
(3, 2) → 3.6
```

В главе 4 (§4.2.2) упоминалось, что компилятор Go позволяет не указывать тип элементов и знак амперсанда при создании литералов врезов, поэтому здесь определение среза `points` в действительности является сокращенной формой инструкции `points := []*Point{&Point{3, 1}, &Point{1, 2}, ...}`.

Хотя здесь и не показано, тем не менее для отображения `distanceForPoint` можно вызывать методы `Delete()`, `Find()` и `Len()`, как было показано в примере с отображением `wordForWord` выше, но передавая первым двум ключи типа `*Point` (потому что функция сравнения принимает значения типа `*Point`, а не `Point`).

Теперь, когда было показано, как использовать упорядоченное отображение, можно перейти к знакомству с его реализацией. Здесь не будут рассматриваться вспомогательные методы и функции, используемые методом `Delete()`, потому что некоторые из них достаточно сложны и их изучение не добавит новых знаний о программировании на языке Go. (Все эти функции можно найти в загружаемых примерах к книге, в файле `qtrac.eu/omap/omap.go`.) Знакомство начнется с представления двух типов, используемых для реализации упорядоченного отображения (`Map` и `node`), и функций-конструкторов. Затем будут рассмотрены методы типа `Map` и некоторые вспомогательные функции. Как это принято при программировании на языке Go, большинство методов очень короткие, и всю основную работу выполняют вспомогательные функции.

```
type Map struct {
    root *node
    less  func(interface{}, interface{}) bool
    length int
}
type node struct {
    key, value interface{}
    red      bool
    left, right *node
}
```

Реализация упорядоченных отображений основана на двух пользовательских типах. Первый тип, `Map`, хранит корень левостороннего красно-черного дерева, функцию для сравнения ключей и количество элементов в отображении. Все поля в этом типе являются неэкспортируемыми, и нулевым значением для поля `less` является значение `nil`, поэтому при создании переменной типа `Map` будет создано недопустимое отображение типа `Map`. Эта особенность отмечена в документации к типу `Map`, где пользователям рекомендуется использовать одну из функций-конструкторов, имеющихся в пакете `omap`.

Второй тип, `node`, представляет единственный элемент *ключ/значение*. Кроме соответствующих полей `key` и `value`, тип `node` имеет три дополнительных поля, необходимых для реализации дерева. Поле `red` типа `bool` используется для идентификации узла как «красного» (`true`) или «черного» (`false`) — оно используется при вращении фрагментов дерева для поддержания сбалансированности. Поля `left` и `right` типа `*node` хранят указатели на левое и правое поддеревья (которые могут иметь значением `nil`).

В пакете `omap` имеются несколько функций-конструкторов. Здесь будет показана универсальная функция `omap.New()` и пара более специализированных.

```
func New(less func(interface{}, interface{}) bool) *Map {  
    return &Map{less: less}  
}
```

Эта универсальная функция может использоваться для создания упорядоченных отображений с ключами и значениями любых встроенных и пользовательских типов, для которых будет предоставлена соответствующая функция сравнения.

```
func NewCaseFoldedKeyed() *Map {  
    return &Map{less: func(a, b interface{}) bool {  
        return strings.ToLower(a.(string)) < strings.ToLower(b.(string))  
    }}  
}
```

Эта функция-конструктор создает пустое упорядоченное отображение со строковыми ключами, которые сравниваются без учета регистра символов.

```
func NewIntKeyed() *Map {  
    return &Map{less: func(a, b interface{}) bool {  
        return a.(int) < b.(int)  
    }}  
}
```

Эта функция-конструктор создает пустое упорядоченное отображение с ключами типа `int`.

В пакете `omap` имеется также функция `omap.NewStringKeyed()` для создания упорядоченных отображений со строковыми ключами, которые сравниваются с учетом регистра символов (ее реализация практически идентична функции `omap.NewCaseFoldedKeyed()`, но в ней отсутствует вызов `strings.ToLower()`), и функция `omap.NewFloat64Keyed()`, действующая подобно функции `omap.NewIntKeyed()`, за исключением того, что в ней используются значения типа `float64` вместо `int`.

```
func (m *Map) Insert(key, value interface{}) (inserted bool) {  
    m.root, inserted = m.insert(m.root, key, value)  
    m.root.red = false
```

```

    if inserted {
        m.length++
    }
    return inserted
}

```

По своей структуре этот метод является типичным для языка Go, так как основную работу он выполняет с помощью вспомогательной функции, в данном случае – неэкспортируемого метода `insert()`. При добавлении новых элементов корень дерева может изменяться либо в результате добавления первого узла в пустое дерево, которое становится его корнем, либо в результате вращений, вовлекающих корень, чтобы сохранить дерево сбалансированным.

Метод `insert()` возвращает корень дерева независимо от того, изменился он или нет, а также логическое значение. Если элемент был добавлен в отображение, возвращается логическое значение `true`, и в этом случае увеличивается длина отображения. Если в логическом значении возвращается `false`, это означает, что элемент с данным ключом уже имеется в отображении, и значение этого элемента было замещено текущим значением `value`, поэтому длина отображения не изменяется. (Здесь не будет объясняться, почему узлы могут быть «красными» или «черными» или почему они могут вращаться. Все это подробно объясняется в статьях Роберта Седжвика (Robert Sedgwick), упоминавшихся в сноске выше.)

```

func (m *Map) insert(root *node, key, value interface{}) (*node, bool) {
    inserted := false
    if root == nil { // Если ключ должен быть вставлен в дерево, то сюда
        return &node{key: key, value: value, red: true}, true
    }
    if isRed(root.left) && isRed(root.right) {
        colorFlip(root)
    }
    if m.less(key, root.key) {
        root.left, inserted = m.insert(root.left, key, value)
    } else if m.less(root.key, key) {
        root.right, inserted = m.insert(root.right, key, value)
    } else { // Ключ уже имеется в дереве, поэтому просто сохранить новое значение
        root.value = value
    }
    if isRed(root.right) && !isRed(root.left) {
        root = rotateLeft(root)
    }
}

```

```
    if isRed(root.left) && isRed(root.left.left) {  
        root = rotateRight(root)  
    }  
    return root, inserted  
}
```

Эта рекурсивная функция выполняет обход дерева в поисках узла с искомым ключом и при необходимости возвращает поддеревья, чтобы обеспечить сбалансированность. Когда метод `Insert()` вызывает этот метод, он передает корень всего дерева (или `nil`, если дерево не имеет узлов), но в последующих рекурсивных вызовах в аргументе `root` передается корень поддерева (который может иметь значение `nil`).

Если новый ключ не будет найден в дереве, в процессе обхода будет достигнут правый крайний его лист для вставки нового ключа – лист со значением `nil`. В этом случае создается новое значение типа `*node` для текущего листа, в котором оба указателя на собственные листы имеют значение `nil`. Здесь не требуется явно инициализировать поля `left` и `right` нового узла (то есть его листы), потому что компилятор Go автоматически присвоит им соответствующие нулевые значения (`nil`), поэтому для инициализации полей структуры с ненулевыми значениями здесь используется синтаксис `ключ: значение`.

Если новый ключ совпадает с одним из ключей в отображении, используется существующий узел и его значение просто замещается новым значением. (Как и в отображениях встроенного типа `map`.) Как следствие этого каждый элемент в упорядоченном отображении будет иметь уникальный ключ.

```
func isRed(root *node) bool { return root != nil && root.red }
```

Эта маленькая вспомогательная функция позволяет узнать, является ли данный узел «красным». Значение `nil` интерпретируется как «черный» узел.

```
func colorFlip(root *node) {  
    root.red = !root.red  
    if root.left != nil {  
        root.left.red = !root.left.red  
    }  
    if root.right != nil {  
        root.right.red = !root.right.red  
    }  
}
```

Эта вспомогательная функция изменяет цвет указанного узла и цвет его листьев на противоположный.

<pre>func rotateLeft(root *node) *node { x := root.right root.right = x.left x.left = root x.red = root.red root.red = true return x }</pre>	<pre>func rotateRight(root *node) *node { x := root.left root.left = x.right x.right = root x.red = root.red root.red = true return x }</pre>
---	--

Эти функции выполняют вращение поддеревьев указанного корня, чтобы поддержать их сбалансированность.

```
func (m *Map) Find(key interface{}) (value interface{}, found bool) {
    root := m.root
    for root != nil {
        if m.less(key, root.key) {
            root = root.left
        } else if m.less(root.key, key) {
            root = root.right
        } else {
            return root.value, true
        }
    }
    return nil, false
}
```

Поскольку данный метод достаточно прост в реализации и вместо рекурсии использует итерации, нет нужды создавать для него вспомогательную функцию.

Метод Find() отыскивает элемент, сравнивая искомый ключ с текущим корневым узлом (в процессе обхода дерева) с помощью функции less(). Делается это с использованием логического тождества $x = y \Leftrightarrow \neg (x < y \vee y < x)$. Она действительна для целых и вещественных чисел, строк, пользовательского типа Point и многих других, но справедлива не для всех типов. При необходимости тип map.Map легко можно было бы расширить, добавив в него отдельную функцию сравнения, возвращающую true в случае равенства сравниваемых ключей.

Обратите внимание, что здесь используются *именованные возвращаемые значения*, хотя им явно ничего не присваивается. Разумеется,

значения им неявно будут присвоены в инструкциях `return`. Именованные значения, как в данном случае, могут пригодиться для нужд документирования функции или метода. Здесь, например, из сигнатуры `Find(key interface{}) (value interface{}, found bool)` видно, что возвращает метод – определить это было бы намного сложнее, если бы метод имел сигнатуру `Find(key interface{}) (interface{}, bool)`.

```
func (m *Map) Delete(key interface{}) (deleted bool) {
    if m.root != nil {
        if m.root, deleted = m.remove(m.root, key); m.root != nil {
            m.root.red = false
        }
    }
    if deleted {
        m.length--
    }
    return deleted
}
```

Удаление элемента из левостороннего красно-черного дерева является достаточно сложной процедурой, поэтому вся основная работа перекладывается на вспомогательный неэкспортируемый метод `remove()` и на вспомогательную функцию, используемую им (здесь не показаны). Если упорядоченное отображение не содержит элементов или если ни один из элементов не соответствует искомому ключу, метод `Delete()` просто ничего не делает и возвращает `false`. Если дерево состоит из единственного элемента и этот элемент требуется удалить, тогда в корень приемника `*omap.Map` сохраняется значение `nil` (и дерево становится пустым). Если удаление было выполнено, длина отображения уменьшается на единицу и вызывающей программе возвращается `true`.

Метод `remove()` отыскивает элемент для удаления, используя то же тождество, что и метод `Find()`.

```
func (m *Map) Do(function func(interface{}, interface{})) {
    do(m.root, function)
}

func do(root *node, function func(interface{}, interface{})) {
    if root != nil {
        do(root.left, function)
        function(root.key, root.value)
        do(root.right, function)
    }
}
```

Метод `Do()` и его вспомогательная функция `do()` используются для обхода всех элементов в упорядоченном отображении, в порядке следования ключей, и вызывают указанную функцию для каждого элемента, передавая его ключ и значение в виде аргументов.

```
func (m *Map) Len() int {  
    return m.length  
}
```

Этот метод просто возвращает длину отображения. Длина увеличивается и уменьшается в методах `omap.Map.Insert()` и `omap.Map.Delete()`, представленных выше.

На этом завершаются обзор пользовательского типа упорядоченных коллекций и знакомство с особенностями объектно-ориентированного программирования на языке Go.

Когда дело доходит до пользовательских типов, для которых любые значения являются допустимыми, можно просто объявить тип (например, в виде структуры) и сделать сам тип и все его поля экспортируемыми (присвоив им имена, начинающиеся с заглавных букв). Этого будет вполне достаточно. (В качестве примеров можно указать типы `image.Point` и `image.Rectangle` в стандартной библиотеке.)

Для пользовательских типов, требующих проверки (например, для составных типов с одним или более полей, где хотя бы одно поле требует проверки присваиваемых ему значений), в языке Go имеется характерная идиома программирования. Поля, требующие проверки, делаются неэкспортируемыми (им присваиваются имена, начинающиеся с маленькой буквы), и реализуются методы доступа к ним.

В случае типов, для которых нулевые значения являются недопустимыми, соответствующие поля делаются неэкспортируемыми, и для них реализуются методы доступа. Факт недопустимости нулевых значений обязательно должен отражаться в документации, и должны предоставляться экспортируемые функции-конструкторы (обычно с именем `New()`). Обычно функции-конструкторы возвращают указатель на значение требуемого типа со всеми полями, заполненными допустимыми значениями.

Значения и указатели на значения с экспортируемыми и неэкспортируемыми полями могут передаваться функциям как обычно, а в случае, если типы реализуют один или более интерфейсов, они, разумеется, могут передаваться как интерфейсы, когда это удобно, то есть когда важно поведение значения, а не его фактический тип.

Очевидно, что программистам с опытом объектно-ориентированного программирования, основанного на наследовании (в таких языках, как C++, Java или Python), придется перестраивать образ своего мышления. Однако удобства динамической типизации и интерфейсы, поддерживаемые в языке Go, а также отсутствие механизма наследования, причиняющего массу неудобств при сопровождении, с лихвой окупят усилия, затраченные на их изучение и освоение. Подход к объектно-ориентированному программированию, используемый в Go, позволяет получить превосходные результаты, если следовать духу языка Go.

6.6. Упражнения

В этой главе предлагается выполнить три упражнения. Первое связано с созданием небольшого, но законченного типа, поля которого требуют проверки. Второе связано с расширением функциональности одного из типов, рассматривавшихся в этой главе. В третьем упражнении потребуются создать небольшой тип коллекций. Первые два упражнения не слишком сложные, но третье упражнение заставит поломать голову.

1. Создайте новый пакет с именем `font` (например, в файле `my_font/font.go`). Цель создания пакета – реализовать тип, представляющий свойства шрифта (такие как название семейства и размер). В пакете должна присутствовать функция `New()`, принимающая название семейства и размер (оба значения должны проверяться) и возвращающая значение типа `*Font` (с допустимыми значениями в неэкспортируемых полях). Также реализуйте методы чтения и записи с проверкой. При проверке следует учитывать следующие требования: имя семейства не должно быть пустой строкой, а размер шрифта должен находиться в диапазоне от 5 до 144 пунктов включительно, при получении недопустимых значений функции должны устанавливать допустимые значения (или сохранять предыдущие) и регистрировать проблему в журнале. Не забудьте добавить метод, реализующий интерфейс `fmt.Stringer`.
Ниже приводится пример создания, изменения и вывода информации о шрифте с помощью пакета.

```
titleFont := font.New("serif", 11)
titleFont.SetFamily("Helvetica")
```

```
titleFont.SetSize(20)
fmt.Println(titleFont)
{font-family: "Helvetica"; font-size: 20pt;}
```

Когда пакет будет готов, скопируйте файл `font/font_test.go` из загружаемых примеров к книге в каталог `my_font` и выполните команду `go test`, чтобы провести простейшее тестирование.

Пример решения можно найти в файле `font/font.go`. Весь пакет занимает примерно 50 строк. В предлагаемом решении метод `String()` возвращает информацию о шрифте в виде стиля CSS (Cascading Style Sheet – каскадные таблицы стилей). В этот пакет достаточно просто, хотя и немного утомительно, можно добавить обработку всех CSS-атрибутов шрифтов, таких как вес, стиль начертания и видоизменения.

2. Скопируйте весь пример `shaper` (иерархическую версию `shaper1`, композиционную версию `shaper2` или версию `shaper3`, какая вам больше нравится, но я рекомендую выбрать `shaper2` или `shaper3`, включая подкаталоги) в новый каталог, например `my_shaper`. Отредактируйте файл `my_shaper/shaper[123].go`: удалите из раздела импортирования все модули, оставив только `image` и `shapes`, и удалите все инструкции в функции `main()`. Отредактируйте файл `my_shaper/shapes/shapes.go`, добавив поддержку новой фигуры с именем `Rectangle`. Эта фигура должна хранить точку рисования, ширину и высоту (все поля, предоставляемые типом `image.Rectangle`), цвет заливки и логический флаг, определяющий необходимость заливки фигуры. Объявление типа `Rectangle` должно быть выполнено в том же стиле, что и другие типы фигур, то есть с неэкспортируемыми полями и интерфейсом (например, иерархическим `RectangularShaper` или композиционными `Rectangler` и `Filleder`) или без интерфейса и экспортируемыми полями (в стиле, характерном для языка Go), и определите API типа. Метод `Draw()` реализовать будет несложно, особенно если воспользоваться неэкспортируемой функцией `drawLine()`, уже имеющейся в пакете `shapes`, и функцией `draw.Draw()`. Не забудьте также добавить в функцию `New()` возможность создания прямоугольников и расширить тип `Option`.

После добавления типа `Rectangle` реализуйте в функции `main()`, в файле `my_shaper/shaper[123].go`, создание и сохранение изображения, представленного на рис. 6.4.

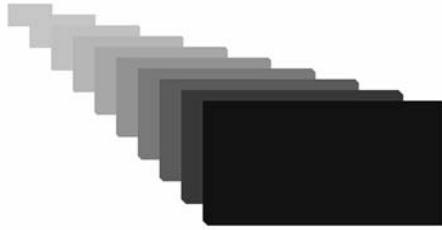


Рис. 6.4. Изображение, созданное с использованием типа *Rectangle*

В загружаемых примерах приводятся три возможных решения. Одно для иерархической версии – в каталоге `shaper_ans1`, одно для композиционной версии – в каталоге `shaper_ans2`, и одно в стиле языка Go – в каталоге `shaper_ans3`. Ниже приводится объявление интерфейса `RectangularShaper` из решения `shaper_ans1`:

```
type RectangularShaper interface {
    Shaper // Fill(); SetFill(); Draw()
    Rect() image.Rectangle
    SetRect(image.Rectangle)
    Filled() bool
    SetFilled(bool)
}
```

В решении `shaper_ans2` имеются два дополнительных интерфейса, `Rectangler` и `Filleder`:

```
type Rectangler interface {
    Rect() image.Rectangle
    SetRect(image.Rectangle)
}
type Filleder interface {
    Filled() bool
    SetFilled(bool)
}
```

В решении, реализованном в стиле, характерном для языка Go, новые интерфейсы создавать не требуется.

В иерархической и композиционной версиях решения используется одно и то же объявление типа `Rectangle`: с неэкспортируемыми полями и методами доступа. Но в решении,

реализованном в стиле языка Go, используются экспортируемые поля, а проверка выполняется только в месте, где они используются.

```
type Rectangle struct {  
    color.Color  
    image.Rectangle  
    Filled bool  
}
```

В файле `shaper_ans1/shapes/shapes.go` объявление типа `Rectangle` и поддерживаемые им методы занимают менее 50 строк. Еще пара строк потребовалась для расширения типа `Option` и пять строк — для расширения функции `New()`. Новая функция `main()` в файле `shaper_ans1/shaper1.go` занимает менее 20 строк. Аналогичные цифры получены в решении `shaper_ans2`. В решении `shaper_ans3` дополнительного программного кода получилось меньше.

Наиболее честолюбивые читатели могут попробовать расширить пример еще больше, реализовав отдельную поддержку цвета для заливки и рамки, где значение `nil` цвета свидетельствует, что заливка или рамка не должна отображаться, а непустое значение определяет цвет заливки или рамки.

3. Создайте в пакете `my_oslice` собственный тип коллекций с именем `Slice`. Этот тип должен реализовывать упорядоченные срезы. Создайте несколько функций-конструкторов, например функцию `New(func(interface{}), interface{}) bool`, принимающую функцию сравнения «меньше чем», и ряд других функций-конструкторов с предопределенными функциями сравнения, таких как `NewStringSlice()` и `NewIntSlice()`. Тип `*oslice.Slice` должен иметь метод `Clear()`, очищающий срез, метод `Add(interface{})`, вставляющий элемент с учетом упорядочения, метод `Remove(interface{}) bool`, удаляющий первое вхождение указанного элемента и сообщаящий об успехе или неудаче, метод `Index(interface{}) int`, возвращающий индекс первого вхождения указанного элемента (или `-1`), метод `At(int) interface{}`, возвращающий элемент в указанной позиции (и возбуждающий аварийную ситуацию в случае выхода индекса за границы среза), и метод `Len() int`, возвращающий количество элементов в срезе.

```
func bisectLeft(slice []interface{}  
    less func(interface{}, interface{}) bool, x interface{}) int {  
    left, right := 0, len(slice)  
    for left < right {  
        middle := int((left + right) / 2)  
        if less(slice[middle], x) {  
            left = middle + 1  
        } else {  
            right = middle  
        }  
    }  
    return left  
}
```

В предлагаемом решении используется функция `bisectLeft()`, которая может оказаться полезной. Если она возвращает значение `len(slice)`, указанный элемент отсутствует в срезе и при добавлении должен быть вставлен в конец. Любое другое значение определяет, что либо указанный элемент присутствует в срезе в данной позиции, либо он отсутствует в срезе и при добавлении должен быть вставлен в данную позицию.

Желающие могут скопировать файл `oslice/oslice_test.go` в каталог `my_oslice`, чтобы протестировать свое решение. Предлагаемое решение находится в файле `oslice/oslice.go` и занимает менее 100 строк. Самым сложным является метод `Add()`, однако функция `InsertStringSlice()` из главы 4 (§4.2.3) может справиться с этой задачей.



7. Параллельное программирование

Параллельное программирование позволяет разработчикам реализовывать многопоточные алгоритмы и писать программы, использующие преимущества многоядерных процессоров и многопроцессорных систем. Недостаток заключается в том, что многопоточные программы намного сложнее в создании, сопровождении и отладке при использовании основных языков программирования (таких как C, C++ и Java). Кроме того, обработку данных не всегда можно разделить между несколькими потоками выполнения. И в любом случае желаемый прирост производительности не всегда достижим из-за накладных расходов, затрачиваемых на организацию самих потоков выполнения, или просто потому, что в многопоточных программах намного проще допустить ошибку.

Одно из решений состоит в том, чтобы вообще отказаться от многопоточной модели выполнения. Например, все тяготы можно переложить на операционную систему, используя ее возможность параллельного выполнения нескольких процессов. Однако при таком подходе придется решать проблемы, связанные с организацией взаимодействий между процессами, а кроме того, накладные расходы на организацию процессов обычно намного выше, чем на организацию потоков выполнения, действующих в общей области памяти.

Язык Go предлагает трехстороннее решение. Во-первых, он предоставляет высокоуровневую поддержку параллельного программирования, что существенно снижает вероятность появления ошибок. Во-вторых, параллельная обработка производится в го-подпрограммах, которые намного легче потоков выполнения. И в-третьих, механизм автоматической сборки мусора освобождает программистов от управления памятью, иногда дьявольски сложного в параллельных программах.

Встроенный в Go высокоуровневый API для создания параллельных программ основан на модели CSP (Communicating Sequential

Processes – взаимодействующие последовательные процессы). Это означает, что явных блокировок и всех хлопот, связанных со своевременным их приобретением и освобождением, можно избежать, а синхронизацию обеспечить приемом и передачей данных через каналы. Это значительно упрощает создание параллельных программ. Кроме того, тогда как десятки потоков выполнения могут создать нагрузку, непомерную для обычного настольного компьютера, тот же самый компьютер успешно справляется с выполнением сотен, тысяч и даже десятков тысяч go-подпрограмм. Подход, используемый в языке Go, позволяет программистам рассуждать в терминах предметной области, а не особенностей использования блокировок и других низкоуровневых механизмов.

Многие языки программирования имеют поддержку низкоуровневых параллельных операций (атомарное сложение, сравнение и присваивание), а также некоторых низкоуровневых механизмов, таких как мьютексы. Но ни в одном из основных языков нет такой высокоуровневой встроенной поддержки параллельного выполнения, как в языке Go (исключение составляют разве что дополнительные библиотеки, которые не являются неотъемлемой частью языка).

В дополнение к высокоуровневой поддержке параллельного выполнения, которая является темой этой главы, в языке Go имеется также поддержка тех же низкоуровневых механизмов, что и в других языках. На самом низком уровне пакет `sync/atomic` из стандартной библиотеки предоставляет функции для выполнения операций атомарного сложения, сравнения и присваивания. Эти дополнительные функции созданы для поддержки реализации поточно-ориентированных алгоритмов синхронизации и структур данных – они не предназначены для использования прикладными программистами. Пакет `sync` реализует самые обычные низкоуровневые примитивы многопоточной модели выполнения: переменные состояния и мьютексы. В большинстве других языков программирования они считаются высокоуровневыми механизмами, поэтому прикладные программисты часто вынуждены использовать их.

При разработке параллельных программ на языке Go прикладные программисты предпочитают использовать высокоуровневые механизмы – каналы и go-подпрограммы. Кроме того, существует тип `sync.Once`, позволяющий обеспечить однократный вызов функции независимо от количества ее вызовов в программе, и тип `sync.WaitGroup`, предоставляющий высокоуровневый механизм синхронизации, как будет показано ниже.

Базовый синтаксис и порядок использования каналов и *go*-подпрограмм уже были представлены в главе 5 (§5.4). Эти сведения не будут повторяться в данной главе, но предполагается их знание, поэтому кому-то может оказаться полезным вернуться и повторно прочитать указанный раздел или хотя бы вскользь пробежаться по нему взглядом, прежде чем продолжить чтение.

Эта глава начинается с обзора некоторых ключевых понятий параллельного программирования на языке Go. Затем в ней будет представлено пять законченных действующих программ, иллюстрирующих особенности параллельного программирования на языке Go, а также некоторые стандартные приемы. Первый пример покажет, как организовать конвейерную обработку, где каждый сегмент конвейера выполняется в отдельной *go*-подпрограмме для обеспечения максимальной пропускной способности. Второй пример покажет, как распределить работу между фиксированным числом *go*-подпрограмм, которые выводят свои результаты независимо друг от друга. Третий пример покажет, как создать поточно-ориентированную структуру данных без использования блокировок или других низкоуровневых примитивов. Четвертый пример покажет три разных варианта, как с использованием фиксированного количества *go*-подпрограмм организовать решение независимых подзадач с объединением результатов. Пятый пример покажет, как создать определенное число *go*-подпрограмм в зависимости от типа обработки и как объединить результаты работы этих *go*-подпрограмм в единый набор результатов.

7.1. Ключевые понятия

Параллельное программирование обычно используется, чтобы распределить обработку данных между одной или несколькими *go*-подпрограммами (выполняющимися параллельно основной *go*-подпрограмме) и вывести результаты по мере их готовности или собрать их воедино для вывода по окончании обработки.

Даже при использовании высокоуровневых механизмов параллельного выполнения, имеющих в языке Go, есть риск попасть в ловушки. Одна из таких ловушек – практически немедленное *завершение* программы до того, как она успеет вывести результаты. Программы на языке Go завершаются автоматически, с завершением главной *go*-подпрограммы, даже если в это время другие *go*-подпрограммы продолжают выполняться. Поэтому необходимо

задержать завершение главной go-подпрограммы, пока вся работа не будет выполнена.

Еще одна ловушка, которой следует опасаться, – *взаимоблокировка*. Одна из разновидностей этой проблемы является практически полной противоположностью первой ловушки: главная и все остальные go-подпрограммы продолжают выполняться даже после завершения обработки данных. Обычно это происходит из-за ошибки передачи сообщения о завершении обработки. Другой разновидностью взаимоблокировки является ситуация, когда две разные go-подпрограммы (или потока выполнения) используют блокировки для защиты доступа к ресурсу и пытаются одновременно приобрести одни и те же блокировки, как показано на рис. 7.1. Проблемы подобного рода могут возникать только при использовании блокировок. Это широко распространенная проблема в других языках программирования, но крайне редкая в Go, потому что приложения на языке Go вместо блокировок могут использовать каналы.

Наиболее типичный способ избежать преждевременного завершения или бессмысленного продолжения работы заключается в том, чтобы заставить главную go-подпрограмму ожидать получения сообщения о завершении работы из канала, созданного исключительно для этой цели (как будет показано чуть ниже, а также в §7.2.2 и §7.2.4). (Для этого можно также использовать специально подготовленное значение, передаваемое как последний «результат», но по сравнению с другими это решение выглядит несколько неуклюже.)

Другой способ избежать ловушек состоит в том, чтобы ждать, пока все go-подпрограммы сообщат о завершении обработки с помощью значения типа `sync.WaitGroup`. Однако использование значения этого типа само может стать причиной взаимоблокировки, особенно если вызвать `sync.WaitGroup.Wait()` в главной go-подпрограмме, когда все go-подпрограммы, выполняющие обработку, окажутся заблокированы (например, в ожидании приема данных из канала). Порядок использования значения типа `sync.WaitGroup` будет описан ниже (§7.2.5).

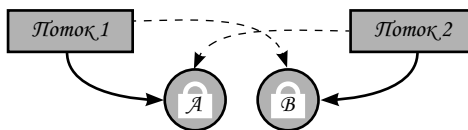


Рис. 7.1. Взаимоблокировка: два или более блокирующих потока пытаются приобрести блокировку, занятую другим потоком

Программы на языке Go могут попасть в состояние взаимоблокировки, даже если в них использовать только каналы и полностью исключить применение блокировок. Например, представьте, что имеется множество go-подпрограмм, которые могут запрашивать друг у друга выполнение некоторых функций (например, отправляя запросы друг другу). Если такая функция пошлет сообщение go-подпрограмме, в рамках которой она выполняется, например чтобы передать некоторые данные, получится взаимоблокировка. Эта ситуация изображена на рис. 7.2. (Позднее будут представлены примеры, где возможны взаимоблокировки такого рода.)

Каналы – это механизм взаимодействий между параллельно выполняющимися go-подпрограммами без применения блокировок. (За кулисами блокировки могут использоваться, но это тонкости реализации, которые нас не касаются.) При использовании каналов синхронизация отправляющего и принимающего концов канала (и соответствующих им go-подпрограмм) выполняется в момент взаимодействия.

По умолчанию каналы являются двунаправленными, то есть они позволяют посылать и принимать значения. Однако очень часто каналы, которые являются полями структур или передаются в виде параметров, являются однонаправленными, то есть они позволяют либо только посылать, либо только принимать значения. В таких случаях семантику использования канала можно выразить (и вынудить компилятор автоматически проверять ее), указывая направление работы канала. Например, тип `chan<- Тип` описывает канал, позволяющий только посылать значения, а тип `<-chan Тип` – только принимать. В предыдущих главах этот синтаксис не использовался, потому что в этом не было необходимости – во всех случаях можно было использовать тип `chan Тип`, – и там рассказывалось совсем о другом. Но с данного момента везде, где это возможно, будут использоваться однонаправленные каналы, потому что они удобнее в использовании и дают преимущество дополнительной проверки на этапе компиляции.

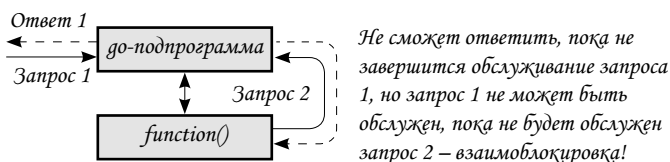


Рис. 7.2. Взаимоблокировка: go-подпрограмма пытается обслужить запрос запросом к себе самой

Посылка через каналы значений таких типов, как `bool`, `int` и `float64`, совершенно безопасна, потому что посылаются их копии, то есть полностью отсутствует риск случайного одновременного обращения к одному и тому же значению из нескольких `go`-подпрограмм. Аналогично посылка строк также безопасна, потому что они являются неизменяемыми значениями.

Посылка *указателей* или *ссылок* (например, срезов или отображений) через каналы по своей природе небезопасна, потому что значения, на которые ссылаются указатели или ссылки, могут изменяться посылающей и принимающей `go`-подпрограммами, что может приводить к непредсказуемым результатам. Поэтому, когда дело доходит до указателей и ссылок, необходимо гарантировать, что обращаться к ним будет только одна `go`-подпрограмма, то есть обеспечить *поочередный доступ*. Исключение составляют случаи, когда в документации явно говорится, что посылать указатель безопасно, например одно и то же значение типа `*regexp.Regexp` может безопасно использоваться несколькими `go`-подпрограммами, потому что ни один из его методов не изменяет состояния значения.

Один из способов обеспечить последовательный доступ заключается в использовании мьютексов. Другой – в соблюдении правила, согласно которому отправитель не должен обращаться к указателю или ссылке после отправки. Это позволит получателю безопасно использовать значение, полученное по указателю или по ссылке, а также посылать указатель или ссылку дальше, при соблюдении того же правила. (Пример использования этого правила будет представлен ниже, в §7.2.4.3.) Недостаток решений на основе подобных правил в том, что они требуют определенной дисциплины программирования. Третий способ обеспечить безопасную работу с указателями и ссылками заключается в реализации экспортируемых методов, которые не изменяют значения, и неэкспортируемых методов, производящих изменения. Такие указатели или ссылки могут передаваться и безопасно использоваться параллельными `go`-подпрограммами посредством экспортируемых методов и позволить использовать неэкспортируемые методы только одной `go`-подпрограмме (например, внутри пакета, где объявляется тип указателя или ссылки; подробнее о пакетах рассказывается в главе 9).

Имеется также возможность посылать через каналы *интерфейсы*, то есть значения, реализующие определенный интерфейс. Значения, реализующие интерфейсы, обеспечивающие доступ только

для чтения, могут безопасно использоваться любым количеством go-подпрограмм (если документация явно не оговаривает иное), но значения, реализующие интерфейсы, которые включают методы для изменения значения, должны обрабатываться подобно указателям – поочередно.

Например, при создании нового изображения вызовом функции `image.NewRGBA()` программа получит значение типа `*image.RGBA`. Этот тип реализует два интерфейса: `image.Image` (содержащий только методы чтения) и `draw.Image` (содержащий все методы интерфейса `image.Image` плюс метод `Set()`). Поэтому одно и то же значение типа `*image.RGBA` безопасно передавать любому количеству go-подпрограмм, если оно посылается как значение, реализующее интерфейс `image.Image`. (К сожалению, безопасность в этом случае может быть нарушена принимающей стороной, использующей операцию *приведения типа*, например к интерфейсу `draw.Image`, поэтому желательно предусмотреть правила, препятствующие появлению таких ситуаций.) А если необходимо послать одно и то же значение типа `*image.RGBA` нескольким go-подпрограммам, которые могут изменить его, его следует посылать как значение типа `*image.RGBA` или как значение, реализующее интерфейс `draw.Image`, и в любом случае необходимо обеспечить последовательный доступ к значению.

Один из простейших способов организации параллельной обработки заключается в использовании одной go-подпрограммы для подготовки заданий и второй – для их выполнения. При таком подходе главной go-подпрограмме остается только создать каналы и обеспечить общую организацию процесса обработки. Например, ниже показано, как в главной go-подпрограмме создать канал для передачи заданий и канал для передачи сообщения о завершении обработки.

```
jobs := make(chan Job)
done := make(chan bool, len(jobList))
```

Здесь создается небуферизованный канал `jobs` для передачи значений пользовательского типа `Job`, а также буферизованный канал `done`, размер буфера которого соответствует количеству заданий в переменной `jobList` типа `[]Job` (инициализация переменной здесь не показана).

После создания каналов и списка заданий можно приступить к обработке.

```
go func() {  
    for _, job := range joblist {  
        jobs <- job // Заблокируется, пока принимающая сторона не прочтает за-  
дание  
    }  
    close(jobs)  
}()
```

В этом фрагменте создается первая go-подпрограмма. Она выполняет итерации по срезу `joblist` и посылает каждое задание в канал `jobs`. Поскольку канал небуферизованный, go-подпрограмма немедленно блокируется и остается в таком состоянии, пока вторая go-подпрограмма не прочтает значение из канала `jobs`. После отправки всех заданий канал `jobs` закрывается, чтобы сообщить принимающей стороне, что заданий больше не будет.

Семантика этого фрагмента далеко не очевидна! Цикл `for` выполняется до исчерпания списка заданий, после чего канал `jobs` закрывается, но все эти действия выполняются параллельно с другими go-подпрограммами. Кроме того, инструкция `go` вернет управление немедленно, оставив программный код выполняться в отдельной go-подпрограмме, и, так как в этот момент никто еще не пытается прочитать задание из канала, go-подпрограмма окажется заблокированной. То есть сразу после выполнения этой инструкции `go` в программе одновременно будут выполняться две go-подпрограммы – главная go-подпрограмма, которая приступит к выполнению следующей инструкции, и эта, вновь созданная go-подпрограмма, заблокированная в ожидании, когда другая go-подпрограмма примет задание из канала. Следовательно, пройдет некоторое время, прежде чем цикл `for` завершится и канал закроется.

```
go func() {  
    for job := range jobs { // Заблокируется в ожидании передачи  
        fmt.Println(job)    // Выполнение задания  
        done <- true  
    }  
}()
```

Этот фрагмент создает вторую go-подпрограмму. Эта go-подпрограмма выполняет итерации по каналу `jobs`, извлекает задания, обрабатывает их (просто выводит) и для каждого задания посылает значение `true` в канал `done`, сообщая о его завершении. (С

таким же успехом можно было бы посылать значение `false`, потому что здесь просто подсчитывается количество посылок в канал `done`, а фактически посланные значения не учитываются.)

Как и первая инструкция `go`, эта инструкция возвращает управление немедленно, а инструкция `for` блокируется в ожидании посылки. То есть в этой точке в программе будут одновременно выполняться уже три `go`-подпрограммы – главная `go`-подпрограмма и две дополнительные `go`-подпрограммы, как показано на рис. 7.3.

Так как первая `go`-подпрограмма уже готова отправить задание, оно будет немедленно получено и обработано второй `go`-подпрограммой. Тем временем первая `go`-подпрограмма снова заблокируется, на этот раз в операции посылки второго задания. Как только вторая `go`-подпрограмма закончит обработку, она пошлет значение в канал `done` – это буферизованный канал, и поэтому операция посылки блокироваться не будет. Затем управление вернется в начало цикла `for` во второй `go`-подпрограмме, и следующее задание будет послано первой и принято второй `go`-подпрограммой и т. д., пока все задания не будут выполнены.

```
for i := 0; i < len(jobList); i++ {  
    <-done // Заблокируется в ожидании передачи  
}
```

Этот заключительный фрагмент начнет выполнение сразу после создания и запуска двух `go`-подпрограмм. Данный программный код выполняется в главной `go`-подпрограмме, и его назначение – гарантировать, что главная `go`-подпрограмма не завершится, пока не будут выполнены все задания.

Цикл `for` выполнит итерации по количеству заданий, но в каждой итерации будет выполнен прием значения из канала `done` (которое тут же отбрасывается), чтобы синхронизировать каждую итерацию с завершением выполнения очередного задания. Если прием из

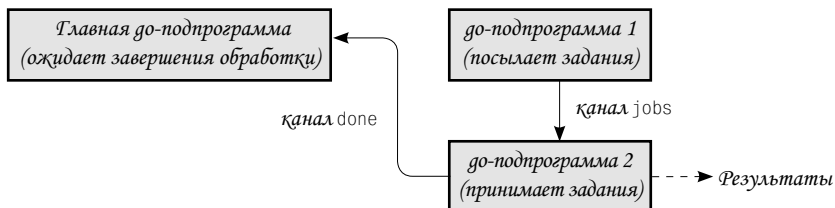


Рис. 7.3. Параллельная независимая подготовка и выполнение заданий

канала невозможен (например, потому что задание еще не выполнено), операция приема будет заблокирована. Как только выполнение заданий завершится, число посылок и приемов из канала `done` станет равно указанному числу итераций и цикл `for` завершится. Теперь главная `go`-подпрограмма сможет завершиться, завершая тем самым программу, и мы можем быть уверены, что все задания были обработаны.

При работе с каналами обычно следует придерживаться двух правил. Во-первых, канал должен закрываться, только если позднее его состояние будет проверяться программой (например, с помощью цикла `for ... range`, инструкцией `select` или операцией контролируемого приема с помощью оператора `<-`). Во-вторых, канал должен закрываться посылающей `go`-подпрограммой, а не принимающей. В общем случае нет смысла закрывать каналы, если потом нигде не будет проверяться их состояние – каналы представляют собой очень легковесный механизм, и они не будут потреблять дополнительных ресурсов, как, например, открытые файлы.

В этом примере по каналу `jobs` выполняются итерации с помощью цикла `for ... range`, поэтому канал закрывается, и делается это внутри посылающей `go`-подпрограммы в соответствии с описанными выше правилами. Канал `done`, напротив, можно не закрывать, потому что ниже в программе нет инструкций, выполнение которых зависело бы от состояния канала.

Этот пример иллюстрирует типичный шаблон параллельного программирования на языке Go, хотя в данном конкретном случае использование параллельно выполняющихся `go`-подпрограмм не несет никаких преимуществ. Некоторые примеры, представленные в следующем разделе, используют похожий шаблон и с выгодой применяют параллельно выполняющиеся `go`-подпрограммы.

7.2. Примеры

Несмотря на небольшое количество синтаксических конструкций поддержки `go`-подпрограмм и каналов в языке Go (`<-`, `chan`, `go`, `select`), их вполне достаточно для реализации параллельных алгоритмов множеством различных способов. Фактически различных способов так много, что невозможно покрыть их все в одной главе. Поэтому мы рассмотрим три шаблона, наиболее часто используемых при создании параллельных программ, – конвейер, множество независимых параллельно выполняемых заданий (с синхронизацией

результатов и без) и множество взаимозависимых параллельно выполняемых заданий – и исследуем конкретные способы их реализации с применением механизмов параллельного выполнения, имеющих в языке Go.

Примеры и упражнения в конце главы позволят получить достаточный объем знаний и навыков параллельного программирования на языке Go, чтобы уверенно применять их при создании новых программ.

7.2.1. Пример: фильтр

Этот первый пример создавался с целью продемонстрировать отдельный шаблон параллельного программирования. Программа легко может быть адаптирована для любой другой работы, где параллельное выполнение может принести выгоду.

Читатели, знакомые с операционной системой Unix, могут заметить, что каналы в языке Go напоминают конвейеры в Unix (за исключением того, что каналы являются двунаправленными, а конвейеры – однонаправленными). Такие конвейеры можно использовать для организации конвейерной обработки, где вывод одной программы подается на ввод другой программы, чей вывод, в свою очередь, подается на ввод третьей программы, и т. д. Например, чтобы получить список всех файлов с исходными текстами на языке Go, находящихся в дереве загружаемых примеров (исключая тестовые файлы), в Unix можно воспользоваться следующим конвейером команд: `find $GOROOT/src -name "*.go" | grep -v test.go`. Одна из замечательных сторон такого подхода состоит в том, что конвейеры легко расширяются. Например, можно добавить `| xargs wc -l`, чтобы для каждого перечисленного файла получить количество строк в нем (плюс общее число строк в конце), и `| sort -n`, чтобы отсортировать файлы по количеству строк в них (в порядке возрастания).

Настоящие конвейеры в стиле Unix можно создавать с помощью функции `io.Pipe()` из стандартной библиотеки. Например, стандартная библиотека Go использует эту функцию для сравнения изображений (см. файл `go/src/pkg/image/png/reader_test.go`).

Помимо функции `io.Pipe()`, создавать конвейеры в стиле Unix можно также с помощью каналов, и этот последний прием будет рассмотрен здесь.

Пример программы `filter` (в файле `filter/filter.go`) принимает несколько аргументов командной строки (например, определяющие

минимальный и максимальный размеры файлов и допустимые расширения имен файлов) и список файлов и выводит имена файлов из списка, соответствующие критериям, заданным в командной строке. Ниже приводится тело функции `main()` программы, состоящее из двух строк.

```
minSize, maxSize, suffixes, files := handleCommandLine()
sink(filterSize(minSize, maxSize, filterSuffixes(suffixes, source(files))))
```

Для обработки аргументов командной строки функция `handleCommandLine()` (здесь не показана) использует пакет `flag` из стандартной библиотеки. Конвейер работает в направлении от самого внутреннего вызова функции (`source(files)`) к самому внешнему (`sink()`). Ниже представлен тот же конвейер, только в более простом для понимания виде.

```
channel1 := source(files)
channel2 := filterSuffixes(suffixes, channel1)
channel3 := filterSize(minSize, maxSize, channel2)
sink(channel3)
```

Функция `source()` принимает срез с именами файлов и возвращает канал типа `chan string`, который присваивается переменной `channel1`. Функция `source()` посылает в канал каждое имя файла по очереди. Далее следуют вызовы двух фильтрующих функций, каждая из которых принимает критерии фильтрации и канал типа `chan string` и возвращает собственный канал типа `chan string`. В этом примере переменной `channel2` присваивается канал, возвращаемый первым фильтром, переменной `channel3` – канал, возвращаемый вторым фильтром. Фильтры выполняют итерации по элементам в канале и посылают каждый элемент, соответствующий их критериям, в канал, возвращаемый ими. Функция `sink()` принимает канал, выполняет итерации по его элементам и выводит их.

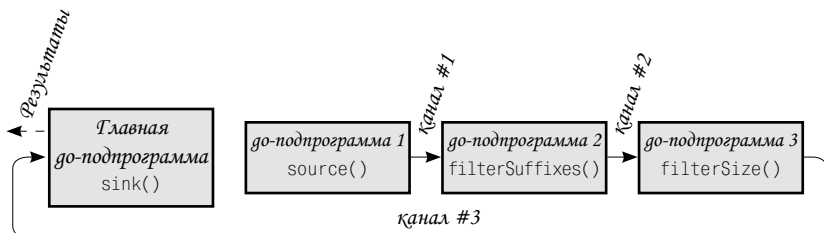


Рис. 7.4. Конвейер до-подпрограмм

На рис. 7.4 схематически изображено, что происходит. В данном случае функция `sink()` выполняется в главной го-подпрограмме, а все остальные функции, составляющие конвейер (`source()`, `filterSuffixes()` и `filterSize()`), выполняются в собственных го-подпрограммах. Это означает, что каждая конвейерная функция возвращает управление немедленно и выполнение быстро достигает функции `sink()`. В этот момент все го-подпрограммы выполняются параллельно, ожидая отправки или приема, пока не будут обработаны все имена файлов.

```
func source(files []string) <-chan string {
    out := make(chan string, 1000)
    go func() {
        for _, filename := range files {
            out <- filename
        }
        close(out)
    }()
    return out
}
```

Эта функция создает канал для передачи имен файлов. Она использует буферизованный канал, поскольку это обеспечивает лучшую производительность в тестах. (Как это нередко бывает, за высокую скорость работы приходится платить повышенным расходом памяти.)

Вслед за каналом создается го-подпрограмма, выполняющая итерации по именам файлов и посылающая каждое имя в канал. Когда все имена файлов будут отправлены, канал закрывается. Как обычно, инструкция `go` возвращает управление немедленно, поэтому между посылкой первого и последнего имен файла и закрытием канала может пройти значительный промежуток времени. Операция посылки в канал не блокируется (по крайней мере, для первой тысячи имен файлов или для всех имен, если их меньше тысячи), но будет блокироваться при посылке большого количества элементов, по крайней мере пока из канала не будет принят один или более элементов.

Как отмечалось выше, по умолчанию каналы создаются двунаправленными, но имеется возможность создать однонаправленный канал. В предыдущем разделе говорилось, что тип `chan<` — тип соответствует каналу, доступному только для отправки, а тип `<-chan`

Тип — доступному только для приема. В конце функция возвращает двунаправленный канал как однонаправленный, доступный только для приема. Разумеется, можно было бы вернуть его как двунаправленный канал, однако использованный способ лучше выражает намерения программиста.

После выполнения инструкции `go`, запускающей анонимную функцию в отдельной `go`-подпрограмме, функция немедленно вернет канал, через который `go`-подпрограмма будет посылать имена файлов. То есть после вызова функции `source()` в программе будут выполняться две `go`-подпрограммы — главная `go`-подпрограмма и дополнительная, созданная в функции.

```
func filterSuffixes(suffixes []string, in <-chan string) <-chan string {
    out := make(chan string, cap(in))
    go func() {
        for filename := range in {
            if len(suffixes) == 0 {
                out <- filename
                continue
            }
            ext := strings.ToLower(filepath.Ext(filename))
            for _, suffix := range suffixes {
                if ext == suffix {
                    out <- filename
                    break
                }
            }
        }
        close(out)
    }()
    return out
}
```

Это первая из двух функций-фильтров и единственная, показанная здесь, потому что функция `filterSize()` имеет практически такую же структуру.

Канал, передаваемый в параметре `in`, может быть двунаправленным или доступным только для приема, но в любом случае объявление типа гарантирует, что внутри функции `filterSuffixes()` он будет использоваться только для приема. (И, как стало понятно после знакомства с возвращаемым значением функции `source()`, канал `in` действительно доступен только для приема.) Соответственно,

здесь, как и в функции `source()`, двунаправленный канал возвращается как доступный только для приема. В обоих случаях можно было бы убрать стрелки `<-`, и функции действовали бы точно так же. Однако добавление оператора направления точнее выражает семантику работы функции и гарантирует, что компилятор строго будет следить за ее соблюдением.

Функция `filterSuffixes()` начинается с создания канала вывода с размером буфера, совпадающим с размером буфера канала ввода, обеспечивая тем самым максимальную пропускную способность. Затем создается `go`-подпрограмма для обработки имен файлов. Внутри `go`-подпрограммы выполняются итерации по элементам в канале `in` (то есть по именам файлов). Если допустимые расширения не были указаны, тогда допустимыми считаются все расширения и принятые имена файлов просто посылаются в канал вывода. Если расширения были указаны и расширение в имени файла, после приведения его к нижнему регистру, совпадает с любым из них, это имя посылается в канал вывода, в противном случае оно просто отбрасывается. (Функция `filepath.Ext()` возвращает расширение имени файла, включая точку в начале, или пустую строку, если расширение в имени отсутствует.)

Как и в функции `source()`, после обработки всех имен файлов канал вывода закрывается, хотя до этого момента может пройти некоторое время. После создания `go`-подпрограммы канал вывода возвращается вызывающей программе, чтобы следующая функция в конвейере смогла принимать из него имена файлов.

В этот момент действуют три `go`-подпрограммы – главная `go`-подпрограмма, `go`-подпрограмма с функцией `source()` и `go`-подпрограмма с данной функцией. А после вызова функции `filterSize()` будут выполняться уже четыре `go`-подпрограммы, причем одновременно.

```
func sink(in <-chan string) {
    for filename := range in {
        fmt.Println(filename)
    }
}
```

Функция `source()` и две функции фильтров действуют в собственных `go`-подпрограммах, взаимодействуя друг с другом посредством каналов. Функция `sink()` выполняется в главной `go`-подпрограмме и

получает данные из последнего канала, созданного второй функцией-фильтром. Она производит итерации по именам файлов, успешно прошедших фильтры (если таковые имеются), и выводит их.

Инструкция `for ... range` в функции `sink()` выполняет итерации по элементам в канале `in`, выводит имена файлов или блокируется операцией приема, пока канал не будет закрыт, это гарантирует, что главная `go`-подпрограмма не завершится, пока не будут обработаны все имена файлов в других `go`-подпрограммах.

Естественно, в конвейер можно добавить и другие функции, для фильтрации имен файлов по дополнительным критериям или обработки имен, прошедших фильтры, главное, чтобы каждая новая функция принимала канал ввода (канал вывода предыдущей функции) и возвращала собственный канал вывода. И конечно, в случае необходимости передавать более сложные значения каналы можно создавать на основе структур, а не только на основе простых строк.

Реализация, представленная в этом подразделе, является отличной иллюстрацией конвейера, однако здесь на каждой стадии выполняется слишком простая обработка данных, чтобы можно было рассчитывать на какие-то выгоды от конвейерного подхода. Выигрыш от применения конвейера будет значительно больше, если на каждой стадии будет выполняться большой объем работ, такой, что каждая `go`-подпрограмма большую часть времени будет занята работой.

7.2.2. Пример: параллельный поиск

Одним из типичных шаблонов параллельного программирования является разделение работы на множество заданий, каждое из которых может выполняться совершенно независимо. Например, на основе этого шаблона в пакете `net/http` стандартной библиотеки реализован HTTP-сервер, который обрабатывает каждый запрос в отдельной `go`-подпрограмме, без взаимодействия с другими `go`-подпрограммами. В этом подразделе будет представлен один из возможных подходов к реализации данного шаблона на примере программы `cgrer` (`concurrent greper` – параллельный поиск).

В отличие от HTTP-сервера из стандартной библиотеки, программа `cgrer` распределяет работу по фиксированному числу `go`-подпрограмм, вместо того чтобы создавать их по мере необходимости. (Пример, использующий переменное число `go`-подпрограмм, будет показан ниже, в §7.2.5.)

Программы `cgrep` принимают регулярное выражение и список имен файлов в виде аргументов командной строки и для каждого совпадения, найденного в файлах, выводят имя файла, номер строки и саму строку. Если совпадений не найдено, программа ничего не выводит.

Программа `cgrep1` (в файле `cgrep1/cgrep.go`) использует три канала, два из которых применяются для отправки и приема структур.

```
type Job struct {
    filename string
    results  chan<- Result
}
```

Эта структура определяет одно задание: имя файла и канал для вывода результатов. Поле `results` можно было бы объявить как `results chan Result`, но, так как при выполнении задания результаты будут только посылаться и никогда не будут приниматься, канал был объявлен однонаправленным, доступным только для отправки.

```
type Result struct {
    filename string
    lino      int
    line      string
}
```

Каждый результат представлен значением этого составного типа и содержит имя файла, номер строки (`lino`) и строку с найденным совпадением.

```
func main() {
    runtime.GOMAXPROCS(runtime.NumCPU()) // Использовать все доступные ядра
    if len(os.Args) < 3 || os.Args[1] == "-h" || os.Args[1] == "--help" {
        fmt.Printf("usage: %s <regex> <files>\n",
            filepath.Base(os.Args[0]))
        os.Exit(1)
    }
    if lineRx, err := regexp.Compile(os.Args[1]); err != nil {
        log.Fatalf("invalid regexp: %s\n", err)
    } else {
        grep(lineRx, commandLineFiles(os.Args[2:]))
    }
}
```

Первая инструкция в функции `main()` сообщает окружению времени выполнения языка Go о необходимости использовать все процессоры (ядра), доступные в системе. Вызов функции `runtime.GOMAXPROCS(0)` возвращает число процессоров и ничего не изменяет. Вызов этой функции с положительным числом определяет количество процессоров, которые должно использовать окружение времени выполнения языка Go. Функция `runtime.NumCPU()` возвращает количество логических процессоров/ядер, имеющихся в системе¹. Эту инструкцию можно добавлять в начало большинства параллельных программ на языке Go, но со временем она станет избыточной, потому что окружение времени выполнения языка Go сможет автоматически адаптироваться под компьютер, на котором она выполняется.

Функция `main()` обрабатывает аргументы командной строки (регулярное выражение и список имен файлов) и вызывает функцию `grep()` для выполнения работы, точнее для управления ею. (Функция `commandLineFiles()` была представлена в главе 4, в §4.4.2.)

Созданное здесь значение `lineRx` (типа `*regexp.Regexp`; §3.6.5) передается функции `grep()` и в конечном счете совместно используется всеми рабочими go-подпрограммами. В общем случае подобное решение должно быть причиной для беспокойства, потому что всегда следует исходить из того, что совместное использование значений, передаваемых по указателю, небезопасно. В таких ситуациях программист сам должен позаботиться о безопасности, например с помощью мьютексов. Как вариант можно пожертвовать небольшим объемом памяти и вместо совместного использования единственного значения создать отдельные значения для каждой go-подпрограммы. К счастью, в данном конкретном случае документация для языка Go явно утверждает, что значения типа `*regexp.Regexp` поддерживают многопоточную модель выполнения. То есть значение типа `*regexp.Regexp` может безопасно использоваться множеством go-подпрограмм.

```
var workers = runtime.NumCPU()
func grep(lineRx *regexp.Regexp, filenames []string) {
    jobs := make(chan Job, workers)
    results := make(chan Result, minimum(1000, len(filenames)))
    done := make(chan struct{}, workers)
```

¹ Некоторые процессоры заявляют, что они имеют большее число ядер, чем в действительности; см. ru.wikipedia.org/wiki/Hyper-threading.

```
go addJobs(jobs, filenames, results) // Выполнить в собственной go-подпрограмме
for i := 0; i < workers; i++ {
    go doJobs(done, lineRx, jobs)    // Каждая в собственной go-подпрограмме
}
go awaitCompletion(done, results)    // Выполнить в собственной go-подпрограмме
processResults(results)              // Заблокируется до конца работы
}
```

Эта функция создает три двунаправленных канала, необходимых программе. Задания распределяются по рабочим go-подпрограммам, количество которых соответствует количеству процессоров, поэтому, чтобы минимизировать ненужные задержки, каналы `jobs` и `done` создаются буферизованными, с размерами буферов, равными этому числу. (Конечно, легко можно было бы добавить параметр командной строки, чтобы дать пользователю возможность определять количество рабочих go-подпрограмм независимо от числа процессоров.) Для канала `results` используется значительно более емкий буфер, как это было сделано в предыдущем подразделе, в примере `filter`, и используется функция `minimum()` (здесь не показана; см. §5.6.1.2, где приводится одна из возможных реализаций, или исходный программный код в файле `cgrep.go`, где находится реализация, используемая здесь).

Вместо того чтобы создать канал `done` типа `chan bool` и не беспокоиться о конкретном посылаемом значении, `true` или `false`, поскольку значение имеет лишь *факт отправки*, здесь создается канал типа `chan struct{}` (то есть *пустая структура*), чтобы точнее выразить семантику использования канала. В канал этого типа можно посылать только пустые структуры (`struct{}{}`), только чтобы обозначить факт отправки, значение которой не играет никакой роли.

После создания каналов вызывается функция `addJobs()`, добавляющая задания в канал `jobs`. Эта функция выполняется в собственной go-подпрограмме. Затем вызывается функция `doJobs()`, выполняющая фактическую работу: в действительности функция вызывается четыре раза, в результате чего запускаются четыре процесса обработки, каждый в своей go-подпрограмме. Затем вызывается функция `awaitCompletion()`, ожидающая, в собственной go-подпрограмме, окончания выполнения всех заданий и затем закрывающая канал `results`. Наконец, вызывается функция `processResults()`, выполняющаяся в главной go-подпрограмме. Эта функция, обрабатывающая результаты, принятые из канала `results`, блокируется до появления результатов в канале и завершается, только когда будут приняты все результаты. Структурная схема программы изображена на рис. 7.5.

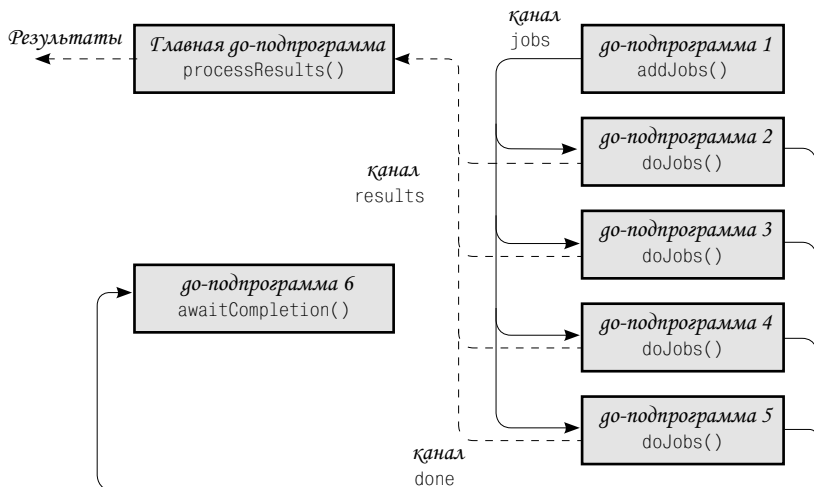


Рис. 7.5. Множество независимых заданий, выполняющихся параллельно

```

func addJobs(jobs chan<- Job, filenames []string, results chan<- Result) {
    for _, filename := range filenames {
        jobs <- Job{filename, results}
    }
    close(jobs)
}

```

Данная функция посылает имена файлов в канал `jobs` (по одному) в составе значений типа `Job`. Канал `jobs` имеет буфер на четыре элемента (по числу рабочих `go`-подпрограмм), поэтому первые четыре задания будут посланы немедленно, а затем `go`-подпрограмма, в которой выполняется функция, окажется заблокированной, в ожидании, пока задание будет принято из канала и освободится место для следующего. Когда все задания будут отправлены, что зависит от количества имен файлов и скорости их обработки, канал `jobs` будет закрыт.

Несмотря на то что два канала, которые передаются функции, в действительности создаются как двунаправленные, в сигнатуре функции они определяются как однонаправленные, доступные только для отправки, потому что именно так канал `jobs` используется в функции и именно так канал `Job.results` объявлен в структуре `Job`.

```
func doJobs(done chan<- struct{}, lineRx *regexp.Regexp, jobs <-chan Job) {  
    for job := range jobs {  
        job.Do(lineRx)  
    }  
    done <- struct{}{}  
}
```

Эта функция вызывается четыре раза, в четырех разных го-подпрограммах, в результате создаются четыре процесса обработки заданий. Каждый процесс выполняет итерации по элементам в общем канале jobs (объявленном как доступный только для приема), и каждый из них блокируется (точнее, блокируется го-подпрограмма, в которой он выполняется), пока из канала не будет принято очередное задание. Для выполнения каждого задания вызывается его метод Job.Do() (который будет показан ниже). Когда все задания будут выполнены, в канал done (объявленный как доступный только для отправки) посылается пустая структура.

В соответствии с соглашениями, принятыми в языке Go, когда принимает параметры с каналами, первыми в сигнатуре объявляются каналы для вывода, а затем каналы для ввода.

```
func awaitCompletion(done <-chan struct{}, results chan Result) {  
    for i := 0; i < workers; i++ {  
        <-done  
    }  
    close(results)  
}
```

Эта функция (наряду с функцией processResults()) гарантирует, что главная го-подпрограмма продолжит выполняться, пока не будут обработаны все задания, благодаря чему исключается вероятность попадания в ловушку, упоминавшуюся в предыдущем разделе (§7.1). Она выполняется в собственной го-подпрограмме и ожидает, пока из канала done не будет принято требуемое количество посылок по числу рабочих го-подпрограмм. Как только цикл завершится, функция закроет канал results, чтобы известить принимающую сторону, что результатов больше не будет. Обратите внимание, что здесь канал results не был объявлен как доступный только для приема (<-chan Result), потому что Go запрещает закрывать такие каналы. Кроме того, функция не закрывает канал done, потому что он нигде не проверяется на закрытие.

```
func processResults(results <-chan Result) {  
    for result := range results {  
        fmt.Printf("%s:%d:%s\n", result.filename, result.lino, result.line)  
    }  
}
```

Эта функция выполняется в главной go-подпрограмме. Она выполняет итерации по элементам в канале `results` или блокируется в ожидании результатов. Как только все результаты будут приняты и обработаны, цикл завершится, функция вернет управление, и программа закончит работу.

Поддержка параллельного выполнения в языке Go реализована настолько гибко, что данный алгоритм — ожидание выполнения заданий, закрытие каналов и вывод результатов — может быть реализован различными способами. Например, программа `cgrep2` (в файле `cgrep2/cgrep.go`), являющаяся разновидностью программы `cgrep1`, обсуждавшейся в этом подразделе, не имеет функций `awaitCompletion()` или `processResults()` и вместо них использует единственную функцию `waitAndProcessResults()`.

```
func waitAndProcessResults(done <-chan struct{}, results <-chan Result) {  
    for working := workers; working > 0; {  
        select { // Блокируется  
        case result := <-results:  
            fmt.Printf("%s:%d:%s\n", result.filename, result.lino,  
                result.line)  
        case <-done:  
            working--  
        }  
    }  
    DONE:  
    for {  
        select { // Не блокируется  
        case result := <-results:  
            fmt.Printf("%s:%d:%s\n", result.filename, result.lino,  
                result.line)  
        default:  
            break DONE  
        }  
    }  
}
```

Функция начинается с цикла `for`, который выполняется, пока существует хотя бы одна активная рабочая go-подпрограмма. Каждый

раз, когда в цикле `for` выполняется инструкция `select`, она блокируется в ожидании приема из канала `results` или `done`. (Если здесь использовать неблокирующую версию инструкции `select`, то есть с разделом `default`, функция просто впустую будет расходовать процессорное время.) Цикл `for` завершается, когда не остается ни одной активной рабочей `go`-подпрограммы, то есть после того, как все рабочие `go`-подпрограммы выполняют посылку значения в канал `done`.

После завершения всех рабочих `go`-подпрограмм начинает выполняться второй цикл `for`. Внутри этого цикла используется неблокируемая инструкция `select`. Если в канале `results` имеются необработанные результаты, выполняется первый раздел `case`, который выводит результат, и выполнение цикла `for` продолжается. Так повторяется, пока не будут выведены все необработанные результаты. Как только выяснится, что в канале нет необработанных результатов (что может произойти в первой же итерации, если к этому моменту канал `results` был пуст), тут же выполняется переход к метке `DONE`. (Простой инструкции `break` без метки недостаточно, так как она прервет выполнение лишь инструкции `select`.) Этот второй цикл не расходует процессорное время впустую, потому что в каждой итерации либо будет выводиться очередной результат, либо цикл будет прерываться, поэтому здесь нет необходимости останавливать выполнение в ожидании очередного результата.

В данном примере функция `waitAndProcessResults()` получилась длиннее и сложнее, чем оригинальные функции `awaitCompletion()` и `processResults()`. Однако решение на основе инструкций `select` может оказаться предпочтительнее, когда приходится обрабатывать несколько разных каналов. Например, благодаря применению инструкции `select` появляется возможность остановить обработку спустя определенный промежуток времени, даже если к этому моменту были обработаны не все результаты.

Ниже приводятся третья и последняя версия, `cgrep3` (в файле `cgrep3/cgrep.go`).

```
func waitAndProcessResults(timeout int64, done <-chan struct{},
    results <-chan Result) {
    finish := time.After(time.Duration(timeout))
    for working := workers; working > 0; {
        select { // Блокируется
        case result := <-results:
            fmt.Printf("%s:%d:%s\n", result.filename, result.lino,
```

```

        result.line)
    case <-finish:
        fmt.Println("timed out")
        return // Время вышло, поэтому завершить с уже имеющимися результатами
    case <-done:
        working--
    }
}
for {
    select { // Не блокируется
    case result := <-results:
        fmt.Printf("%s:%d:%s\n", result.filename, result.lino,
            result.line)
    case <-finish:
        fmt.Println("timed out")
        return // Время вышло, поэтому завершить с уже имеющимися результатами
    default:
        return
    }
}
}

```

Это разновидность одноименной функции из примера `cgrep2`, отличающейся дополнительным параметром `timeout`. Функция `time.After()` принимает значение `time.Duration` (фактически – число наносекунд) и возвращает канал, в который она посылает текущее время *по истечении* указанного интервала `time.Duration`. Здесь возвращаемый ею канал присваивается переменной `finish` и в обе инструкции `select` включается раздел для приема значения из этого канала. По истечении установленного интервала времени (то есть после отправки значения в канал `finish`) функция возвращает управление, и программа завершается, даже если работа еще не закончена.

Если все результаты будут собраны до истечения указанного интервала времени (то есть все рабочие `go`-подпрограммы успеют закончить работу), выполнение первого цикла `for` завершится, и начнется второй цикл `for`, такой же, как в примере `cgrep2`. Единственное отличие здесь заключается в том, что во второй инструкции `select` присутствует раздел `case` для обработки канала `finish` и вместо прерывания второго цикла `for` в разделе `default` просто выполняется возврат.

Теперь, когда было показано, как организовать параллельную обработку заданий, можно завершить обзор примеров `cgrep` знакомством с методом, реализующим обработку одного задания.

```
func (job Job) Do(lineRx *regexp.Regexp) {
    file, err := os.Open(job.filename)
    if err != nil {
        log.Printf("error: %s\n", err)
        return
    }
    defer file.Close()
    reader := bufio.NewReader(file)
    for lino := 1; ; lino++ {
        line, err := reader.ReadBytes('\n')
        line = bytes.TrimRight(line, "\n\r")
        if lineRx.Match(line) {
            job.results <- Result{job.filename, lino, string(line)}
        }
        if err != nil {
            if err != io.EOF {
                log.Printf("error:%d: %s\n", lino, err)
            }
            break
        }
    }
}
```

Этот метод используется для обработки одного файла. Он получает указатель `*regexp.Regexp`, который, что вообще необычно для указателей, безопасно можно использовать в многопоточном окружении, поэтому не имеет никакого значения, как много го-подпрограмм будут использовать его. Практически весь программный код в этом методе должен быть вам понятен: он открывает файл для чтения, обрабатывает возможную ошибку и, если ошибка не возникла, откладывает закрытие файла до завершения. Затем создает буферизованное значение для чтения из файла, что упрощает итерации по строкам. Всякий раз, когда обнаруживается совпадение с регулярным выражением, в канал `results` посылается значение типа `Result`: операция отправки будет блокироваться после заполнения буфера канала. Для любого обрабатываемого файла может быть произведено произвольное количество результатов, в том числе и нуль, если ни в одной из строк не будет обнаружено совпадения с регулярным выражением.

Как это принято в языке Go при работе с текстовыми файлами, если при чтении строки возникла ошибка, она обрабатывается после обработки самой строки. Если метод `bufio.Reader.ReadBytes()`

столкнется с ошибкой (включая конец файла), он вернет байты, которые удалось прочесть до ошибки вместе с самой ошибкой. Иногда последняя строка в файле не заканчивается символом перевода строки, поэтому, чтобы гарантировать обработку последней строки в файле (не содержащей символ перевода строки), ошибка обрабатывается после обработки строки. Недостаток такого способа обработки ошибок – в том, что если регулярное выражение может совпасть с пустой строкой и будет получено не `nil` и не `io.EOF` значение ошибки, будет найдено ложное совпадение. (Разумеется, эту ситуацию вполне можно обработать.)

Метод `bufio.Reader.ReadBytes()` возвращает байты, прочитанные до указанного (включительно) байта (или до конца файла, если указанный байт не будет встречен). Символ перевода строки здесь не нужен, поэтому он удаляется вызовом метода `bytes.TrimRight()`, который удаляет указанный символ или символы с правого конца строки. (Так же, как функция `strings.TrimRight()`; см. табл. 3.7.) Чтобы сделать программу кросс-платформенной, удаляются оба символа, перевод строки и возврат каретки.

Обратите также внимание, что строки читаются как срезы с байтами и сопоставление выполняется с помощью метода `regexp.Regexp.Match()`, а не `regexp.Regexp.MatchString()`. Поэтому для совпадающих строк необходимо выполнить (очень недорогое) преобразование значения типа `[]byte` в значение типа `string`. Кроме того, для удобства подсчет строк начинается с единицы, а не с нуля.

Особенно интересным аспектом архитектуры программы `cgrep` являются простота и независимость организации параллельной обработки от фактической реализации выполнения заданий (в методе `Job.Do()`), где единственным связующим звеном является канал `results`. Такое разделение вообще характерно для параллельных программ на языке Go и выглядит предпочтительнее, по сравнению с использованием низкоуровневых конструкций (таких как мьютексы), где инструкции приобретения и освобождения блокировок разбросаны по всей программе и могут загромождать и усложнять логику работы программы.

7.2.3. Пример: поточно-ориентированное отображение

Пакеты `sync` и `sync/atomic` из стандартной библиотеки предоставляют низкоуровневые операции, необходимые для реализации

поточно-ориентированных алгоритмов и структур данных. Однако точно так же можно взять за основу имеющуюся структуру данных, такую как отображение или срез (или `omap.Map`; §6.5.3), и превратить ее в поточно-ориентированную, снабдив механизм на основе высокоуровневых каналов, гарантирующим *очередность операций доступа*.

В этом подразделе будет создано поточно-ориентированное отображение с ключами типа `string` и значениями типа `interface{}` (то есть *любыми* значениями), которое может совместно использоваться множеством go-подпрограмм без применения блокировок. (Конечно, если сохранять в отображении указатели или ссылки на значения, они должны интерпретироваться как доступные только для чтения, или следует предусмотреть механизм, гарантирующий очередность операций доступа к ним.) Реализация поточно-ориентированного отображения находится в файле `safemap/safemap.go` и содержит экспортируемый интерфейс `SafeMap`, определяющий методы обеспечения безопасного доступа к отображению, и неэкспортируемый конкретный тип `safeMap`, реализующий этот интерфейс. Практическое применение поточно-ориентированного отображения будет продемонстрировано в следующем подразделе (§7.2.4).

Безопасность отображения обеспечивается неэкспортируемым методом, заключающим отображение в go-подпрограмму. Единственный способ получить доступ к отображению – использовать каналы, и этого уже достаточно, чтобы обеспечить очередность операций доступа. Метод выполняет бесконечный цикл, извлекая значения из канала ввода, который блокируется в ожидании команд (таких как «вставить это значение», «удалить этот элемент» и т. д.).

Для начала познакомимся с интерфейсом `SafeMap`, затем с экспортируемыми методами типа `safeMap`, далее с функцией `New()` из пакета `safemap` и, наконец, с неэкспортируемым методом `safeMap.run()`.

```
type SafeMap interface {
    Insert(string, interface{})
    Delete(string)
    Find(string) (interface{}, bool)
    Len() int
    Update(string, UpdateFunc)
    Close() map[string]interface{}
}

type UpdateFunc func(interface{}, bool) interface{}
```

Все эти методы реализуются типом `safeMap`. (Шаблон использования экспортируемого интерфейса и неэкспортируемого конкретного типа рассматривался в предыдущей главе.)

Тип `UpdateFunc` определяет сигнатуру функции обновления: она будет рассматриваться после знакомства с методом `Update()` ниже.

```

type safeMap chan commandData
type commandData struct {
    action  commandAction
    key     string
    value   interface{}
    result  chan<- interface{}
    data    chan<- map[string]interface{}
    updater UpdateFunc
}
type commandAction int
const (
    remove commandAction = iota
    end
    find
    insert
    length
    update
)

```

Тип `safeMap` основан на канале, посредством которого можно посылать и принимать значения типа `commandData`. Каждое значение `commandData` определяет операцию и данные, участвующие в этой операции, например большинство методов требуют ключа для идентификации элемента. Подробнее все поля будут описываться при знакомстве с методами типа `safeMap`.

Обратите внимание, что оба канала, `result` и `data`, объявлены как однонаправленные. Иными словами, поточно-ориентированное отображение само может посылать значения в них, но не может принимать значения из них. Как будет показано ниже, методы, создающие эти каналы, создают их двунаправленными, благодаря чему эти каналы могут использоваться для приема всего, что в них посылает поточно-ориентированное отображение.

```

func (sm safeMap) Insert(key string, value interface{}) {
    sm <- commandData{action: insert, key: key, value: value}
}

```

Этот метод является безопасным эквивалентом инструкции `m[key] = value`, где `m` — значение типа `map[string] interface{}`. Он создает значение типа `commandData` с командой `insert`, указанным ключом `key` и значением `value`, и посылает его поточно-ориентированному отображению, которое, как было показано выше, имеет тип `chan commandData`. (Порядок создания значений составных типов, когда инициализируются лишь отдельные поля, был показан ранее, в §6.4.)

При обзоре функции `New()` из пакета `safemap` будет показано, что значение типа `safeMap` возвращается функцией `New()` (как интерфейс `SafeMap`) уже связанным с `go`-подпрограммой. Метод `safeMap.run()` выполняется в `go`-подпрограмме и образует замыкание, заключающее канал `safeMap`. Метод также содержит в себе фактическое отображение, используемое для хранения элементов, и цикл `for`, который производит итерации по элементам в канале `safeMap` и выполняет команды, принимаемые из канала.

```
func (sm safeMap) Delete(key string) {
    sm <- commandData{action: remove, key: key}
}
```

Этот метод посылает команду на удаление элемента с указанным ключом.

```
type findResult struct {
    value interface{ }
    found bool
}

func (sm safeMap) Find(key string) (value interface{ }, found bool) {
    reply := make(chan interface{ })
    sm <- commandData{action: find, key: key, result: reply}
    result := (<-reply).(findResult)
    return result.value, result.found
}
```

Метод `safeMap.Find()` создает собственный канал `reply`, чтобы получить ответ от поточно-ориентированного отображения, которое может только посылать данные в канал `reply`, и затем посылает поточно-ориентированному отображению команду `find` с указанным ключом и собственным каналом `reply`. Поскольку ни один из каналов не имеет буфера, операция отправки данных блокируется, пока поточно-ориентированное отображение не обработает запросы от

других go-подпрограмм. После отправки команды метод немедленно получает ответ (в виде структуры `findResult` для команды `find`), элементы которого возвращаются вызывающей программе. Именованные возвращаемые значения здесь используются, только чтобы сделать их назначение более очевидным.

```
func (sm safeMap) Len() int {
    reply := make(chan interface{})
    sm <- commandData{action: length, result: reply}
    return (<-reply).(int)
}
```

Этот метод имеет такую же структуру, что и метод `Find()`, — он создает и посылает собственный канал `reply` и передает полученный ответ вызывающей программе.

```
func (sm safeMap) Update(key string, updater UpdateFunc) {
    sm <- commandData{action: update, key: key, updater: updater}
}
```

На первый взгляд, имя этого метода кажется несоответствующим его сигнатуре, согласно которой во втором аргументе он принимает функцию типа `func(interface{}, bool) interface{}` (см. выше). Метод посылает поточно-ориентированному отображению команду `update` с ключом `key` и функцией `updater`. При выполнении команды будет вызвана функция `updater` со значением элемента, соответствующего указанному ключу (или `nil`, если такой элемент отсутствует), и значением типа `bool`, указывающее на присутствие элемента. Элементу будет присвоено значение, возвращаемое функцией `updater` (при необходимости будет создан новый элемент с указанным ключом `key` и значением, полученным от функции `updater`).

Важно отметить, что если функция `updater` вызовет метод типа `safeMap`, возникнет ситуация взаимоблокировки! Причины этого будут описываться при знакомстве с методом `safeMap.safeMap.run()` ниже.

Но зачем нужен этот странный метод и как им пользоваться?

Когда необходимо вставить, удалить или отыскать элемент в поточно-ориентированном отображении, можно воспользоваться методами `Insert()`, `Delete()` и `Find()`. Но как быть, если необходимо изменить значение имеющегося элемента? Например, представьте, что

отображение используется для хранения цен на различные товары и необходимо увеличить цену некоторого товара на 5%. При использовании обычного отображения можно просто записать `m[key] *= 1.05`. В этом случае, если элемент с ключом `key` существует, его значение будет увеличено на 5%, в противном случае, благодаря автоматической инициализации новых переменных нулевыми значениями, будет создан новый элемент с ключом `key` и с нулевым значением. Ниже показано, как можно попытаться реализовать то же самое при использовании поточно-ориентированного отображения, хранящего значения типа `float64`.

```
if price, found := priceMap.Find(part); found { // ОШИБКА!  
    priceMap.Insert(part, price.(float64)*1.05)  
}
```

Проблема здесь в том, что одно и то же отображение `priceMap` может совместно использоваться несколькими го-подпрограммами, которые могут одновременно попытаться изменить его содержимое между вызовами методов `Find()` и `Insert()` в этом фрагменте. Поэтому нет никакой гарантии, что к моменту вызова метода `Insert()` новая цена действительно будет на 5% больше цены, хранящейся в этот момент в отображении.

Подобные операции должны выполняться *атомарно*, то есть извлечение и изменение значения должны выполняться как единая, непрерываемая операция. Именно это и обеспечивает метод `Update()`.

```
priceMap.Update(part, func(price interface{}, found bool) interface{} {  
    if found {  
        return price.(float64) * 1.05  
    }  
    return 0.0  
})
```

Этот фрагмент демонстрирует, как выполняется атомарное обновление. Если элемент с указанным ключом отсутствует, будет создан новый элемент со значением 0.0. В противном случае значение существующего элемента будет увеличено на 5%. Поскольку обновление в ответ на команду `update` выполняется в рамках го-подпрограммы поточно-ориентированного отображения, никакие другие команды (например, из другой го-подпрограммы) не могут быть выполнены в это же время.

```
func (sm safeMap) Close() map[string]interface{} {  
    reply := make(chan map[string]interface{})  
    sm <- commandData{action: end, data: reply}  
    return <-reply  
}
```

Метод `Close()` действует подобно методам `Find()` и `Len()`, но преследует две разные цели. Во-первых, он закрывает канал типа `safeMap` (внутри метода `safeMap.run()`), что делает невозможным дальнейшее изменение отображения. Это вызывает завершение цикла `for ... range` в методе `safeMap.run()` и тем самым освобождает `go`-подпрограмму для утилизации сборщиком мусора. Во-вторых, он возвращает фактическое отображение `map[string]interface{}{}`, которое вызывающая программа может сохранить или отбросить. Метод `Close()` может быть вызван только один раз, независимо от количества `go`-подпрограмм, обращающихся к отображению, и после его вызова никакой другой метод не может быть вызван. Это означает, что если сохранить полученное отображение, к нему можно свободно обращаться как к обычному отображению (то есть в единственной `go`-подпрограмме).

На этом завершается обзор экспортируемых методов типа `safeMap`. Осталось лишь познакомиться с функцией `New()` в пакете `safemap`, создающей и возвращающей значение типа `safeMap` как интерфейс `SafeMap` для использования за пределами пакета, и с методом `safeMap.run()`, хранящим канал, предоставляющим значение типа `map[string]interface{}{}` для хранения данных и обрабатывающим все команды.

```
func New() SafeMap {  
    sm := make(safeMap) // тип safeMap chan commandData  
    go sm.run()  
    return sm  
}
```

`safeMap` — это тип `chan commandData`, поэтому для создания канала и получения ссылки на него необходимо использовать встроенную функцию `make()`. После создания поточно-ориентированного отображения вызывается его неэкспортируемый метод `run()`, создающий фактическое отображение и принимающий команды. Метод `run()` выполняется в собственной `go`-подпрограмме, и, как обычно, инструкция `go` возвращает управление немедленно. В конце функция `New()` возвращает значение типа `safeMap` как интерфейс `SafeMap`.

```
func (sm safeMap) run() {  
    store := make(map[string]interface{})  
    for command := range sm {  
        switch command.action {  
        case insert:  
            store[command.key] = command.value  
        case remove:  
            delete(store, command.key)  
        case find:  
            value, found := store[command.key]  
            command.result <- findResult{value, found}  
        case length:  
            command.result <- len(store)  
        case update:  
            value, found := store[command.key]  
            store[command.key] = command.updater(value, found)  
        case end:  
            close(sm)  
            command.data <- store  
        }  
    }  
}
```

После создания фактического отображения метод `run()` начинает выполнять бесконечный цикл, производящий итерации по элементам канала, блокируясь в моменты, когда в канале отсутствуют какие-либо команды.

Поскольку отображение `store` – не более чем обычное отображение, все операции, выполняемые в ответ на получение соответствующих команд, должны быть понятны. Единственный сложный момент здесь – это обработка команды `update`, где значению элемента присваивается значение, возвращаемое функцией `command.updater()` (описанной выше). Раздел `end` соответствует вызову метода `Close()`. Здесь сначала закрывается канал для предотвращения приема дальнейших команд, и затем фактическое отображение посылается обратно.

Обратите внимание, что если функция `command.updater()` выполнит вызов метода типа `safeMap`, возникнет ситуация взаимоблокировки. Это обусловлено тем, что обработка команды `update` не может завершиться, пока `command.updater()` не вернет управление, но если функция вызовет метод типа `safeMap`, этот вызов заблокируется в ожидании завершения обработки текущей команды, то есть ни то,

ни другое не смогут завершиться. Эта разновидность взаимоблокировок изображена на рис. 7.2 выше.

Очевидно, что использование поточно-ориентированного отображения сопряжено с дополнительными накладными расходами в сравнении с обычным отображением. Для выполнения каждой команды требуется создать структуру `commandData` и послать ее в канал, при этом команды, которые могут посылаться сколь угодно большим количеством `go`-подпрограмм, выполняются строго поочередно. Одно из альтернативных решений заключается в использовании обычного отображения, доступ к которому защищается с помощью значения типа `sync.Mutex` или `sync.RWMutex`. Другое альтернативное решение состоит в создании собственной поточно-ориентированной структуры, подобной тем, что описываются в литературе (например, см. «The Art of Multiprocessor Programming» в приложении С ниже). Еще одно альтернативное решение заключается в том, чтобы в каждой `go`-подпрограмме создавать собственное отображение, дабы полностью исключить необходимость синхронизации, и в конце объединять их. Тем не менее поточно-ориентированное отображение, описанное здесь, удобно в использовании и может пригодиться во многих ситуациях. Практическое использование поточно-ориентированного отображения будет демонстрироваться в следующем подразделе наряду с парой альтернативных решений, для сравнения.

7.2.4. Пример: отчет о работе веб-сервера

Часто в программах, реализующих параллельную обработку данных, возникает необходимость обновить некоторую совместно используемую структуру. Обычно для обеспечения очередности доступа к такой структуре данных используются мьютексы. В языке `Go` в подобных ситуациях можно использовать мьютексы или каналы. В данном подразделе сначала будет продемонстрирован подход на основе каналов и поточно-ориентированного отображения, созданного в предыдущем подразделе. Затем будет показан способ достижения того же самого с помощью обычного отображения, доступ к которому защищен мьютексом. И наконец, будет представлен прием с использованием локальных независимых отображений, не требующих разграничения доступа, позволяющий обеспечить максимальную пропускную способность, и с использованием каналов, применяемых для обновления общего отображения в конце.

Все версии программы `apachereport` делают одно и то же: они читают файл `access.log`, производимый веб-сервером `Apache`, и

выводят количество обращений к каждой HTML-странице, зафиксированных в этом файле журнала. Эти файлы журналов могут иметь огромный размер, поэтому здесь будут использоваться несколько го-подпрограмм: одна – для чтения строк из файла и три – для обработки строк. При первом обнаружении каждая HTML-страница должна добавляться в отображение со значением счетчика, равным 1, и при каждом последующем обнаружении счетчик должен увеличиваться. То есть, несмотря на то что го-подпрограммы обрабатывают строки из файла журнала независимо друг от друга, все они должны обновлять одно и то же отображение. Каждая версия программы использует свой способ обновления отображения.

7.2.4.1. Синхронизация с применением разделяемого поточно-ориентированного отображения

В этом подразделе будет представлена программа `apachereport1` (в файле `apachereport1/apachereport.go`). Она использует поточно-ориентированное отображение, разработанное в предыдущем подразделе в качестве совместно используемого отображения. Структура параллельной обработки в программе изображена на рис. 7.6 (ниже).

На рис. 7.6 го-подпрограмма 2 используется для наполнения рабочего канала строками из файла журнала, а го-подпрограммы с 3 по 5 обрабатывают строки и обновляют отображение типа `safeMap`. Непосредственные операции с отображением `safeMap` выполняются в отдельной го-подпрограмме, то есть всего в программе используются шесть го-подпрограмм.

```
var workers = runtime.NumCPU()
func main() {
    runtime.GOMAXPROCS(runtime.NumCPU()) // Использовать все доступные ядра
    if len(os.Args) != 2 || os.Args[1] == "-h" || os.Args[1] == "--help" {
        fmt.Printf("usage: %s <file.log>\n", filepath.Base(os.Args[0]))
        os.Exit(1)
    }
    lines := make(chan string, workers*4)
    done := make(chan struct{}, workers)
    pageMap := safemap.New()
    go readLines(os.Args[1], lines)
    processLines(done, pageMap, lines)
    waitUntil(done)
    showResults(pageMap)
}
```

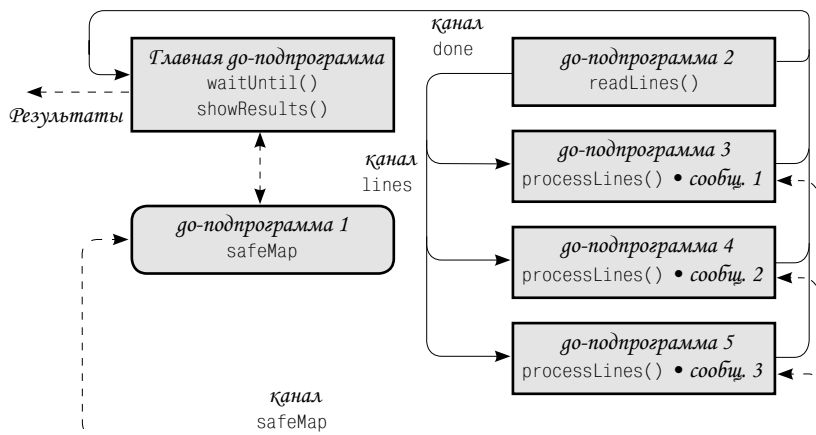


Рис. 7.6. Множество взаимонезависимых заданий с синхронизацией результатов

Функция `main()` начинается с инструкции, сообщающей окружению времени выполнения языка Go о необходимости использовать все доступные процессоры. Затем она создает два канала, необходимые для организации обработки. Канал `lines` используется для отправки строк из файла журнала, которые будут извлекаться рабочими `go`-подпрограммами. Каналу `lines` придается небольшой буфер, чтобы уменьшить вероятность простоя рабочих `go`-подпрограмм в ожидании поступления новых строк. Канал `done` используется для слежения за завершением работы, и поскольку интерес здесь представляет лишь сам факт отправки в канал, в качестве значений для этого канала используются пустые структуры. Канал `done` буферизован, чтобы не блокировать `go`-подпрограмм, желающих сообщить о завершении работы.

Вслед за каналами создается новое значение неэкспортируемого типа `safeMap`, возвращаемое функцией `safemap.New()` в виде экспортируемого интерфейса `SafeMap`, который свободно может передаваться между функциями. Затем запускается `go`-подпрограмма чтения строк из файлов и следом за ней – `go`-подпрограммы обработки строк. Затем функция ожидает завершения всех рабочих `go`-подпрограмм и выводит результаты.

```

func readLines(filename string, lines chan<- string) {
    file, err := os.Open(filename)
    if err != nil {

```

```

    log.Fatal("failed to open the file:", err)
}
defer file.Close()
reader := bufio.NewReader(file)
for {
    line, err := reader.ReadString('\n')
    if line != "" {
        lines <- line
    }
    if err != nil {
        if err != io.EOF {
            log.Println("failed to finish reading the file:", err)
        }
        break
    }
}
close(lines)
}

```

Эта функция очень похожа на функции, встречавшиеся выше. Первая важная особенность — каждая строка посылается в канал `lines`, доступный только для отправки, при этом, если буфер канала полон, операция отправки блокируется, пока другая го-подпрограмма не примет строку из канала. Естественно, заблокирована будет только данная го-подпрограмма, все остальные продолжат работу, как ни в чем не бывало. Вторая важная особенность — после отправки всех строк канал `lines` закрывается, чтобы известить предполагаемые принимающие го-подпрограммы об исчерпании данных. Однако имейте в виду, что данная го-подпрограмма выполняется параллельно с другими го-подпрограммами, в частности с рабочими го-подпрограммами, обрабатывающими строки, поэтому инструкция с вызовом функции `close()` будет достигнута, только когда большая часть работы будет уже выполнена.

```

func processLines(done chan<- struct{}, pageMap safemap.SafeMap,
    lines <-chan string) {
    getRx := regexp.MustCompile(`GET[ \t]+([^\t\n]+[.]\html?)`)
    incrementer := func(value interface{}, found bool) interface{} {
        if found {
            return value.(int) + 1
        }
        return 1
    }
}

```

```
for i := 0; i < workers; i++ {
    go func() {
        for line := range lines {
            if matches := getRx.FindStringSubmatch(line);
                matches != nil {
                pageMap.Update(matches[1], incrementer)
            }
        }
        done <- struct{}{}
    }()
}
```

Эта функция следует соглашению о порядке следования аргументов, согласно которому первым следует канал вывода (в данном случае канал `done`), а затем канал ввода (в данном случае канал `lines`).

Функция создает `go`-подпрограммы (три в этом примере), выполняющие обработку строк. Каждая `go`-подпрограмма использует одно и то же значение типа `*regexp.Regexp` (это значение в документации описывается как допускающее возможность использования в многопоточном окружении, что в общем случае необычно для указателей) и одну и ту же функцию `incrementer()` (так как она не имеет побочных эффектов и не обращается к совместно используемым данным). Они также совместно используют одно и то же значение `pageMap` (реализующее интерфейс `SafeMap`), поскольку даже методы, изменяющие фактическое значение, могут безопасно использоваться в многопоточном окружении.

Функция `regexp.Regexp.FindStringSubmatch()` возвращает `nil` в случае отсутствия совпадений или срез типа `[]string`, первый элемент которого содержит совпадение со всем регулярным выражением, а каждый последующий – совпадения с соответствующими подвыражениями в круглых скобках. В данном случае имеется одно такое подвыражение, то есть при обнаружении совпадения срез будет содержать точно два элемента: текст совпадения со всем регулярным выражением и текст совпадения с подвыражением в круглых скобках, в данном случае – имя файла HTML-страницы.

Каждая `go`-подпрограмма принимает разные строки из канала `lines`, доступного только для приема, которые извлекаются из файла `go`-подпрограммой, созданной в функции `readLines()`. Если строка соответствует регулярному выражению, отыскивающему GET-запросы к HTML-файлам, вызывается метод `safeMap.Update()`

с именем файла страницы (`matches[1]`) и функцией `incrementer()`. Функция `incrementer()` выполняется внутри `go`-подпрограммы поточно-ориентированного отображения. Она возвращает увеличенные значения счетчиков для существующих страниц и значение 1 для страниц, отсутствующих в отображении. (В предыдущем подразделе говорилось, что если функция, передаваемая методу `safeMap.Update()`, попытается вызвать какой-либо другой метод интерфейса `SafeMap`, возникнет состояние взаимоблокировки.) После обработки всех строк каждая рабочая `go`-подпрограмма посылает пустую структуру в канал `done`, доступный только для отправки, чтобы сообщить о завершении работы.

```
func waitUntil(done <-chan struct{}) {
    for i := 0; i < workers; i++ {
        <-done
    }
}
```

Эта функция выполняется в главной `go`-подпрограмме и блокируется в операции приема из канала `done`, доступного только для приема. Когда все рабочие `go`-подпрограммы пошлют пустые структуры в канал `done`, цикл `for` завершится. Как обычно, здесь нет необходимости беспокоиться о закрытии канала `done`, потому что ни одна функция в программе не проверяет его закрытие. Благодаря блокировке в операции приема эта функция гарантирует, что обработка закончится до того, как завершится главная `go`-подпрограмма.

```
func showResults(pageMap safemap.SafeMap) {
    pages := pageMap.Close()
    for page, count := range pages {
        fmt.Printf("%8d %s\n", count, page)
    }
}
```

Когда все строки будут прочитаны и все совпадения будут добавлены в отображение, будет вызвана эта функция для вывода результатов. Она начинается с вызова метода `safemap.SafeMap.Close()`, который закроет канал поточно-ориентированного отображения, завершит выполнение метода `safeMap.run()` в отдельной `go`-подпрограмме и вернет фактическое отображение типа `map[string]interface{}`. После этого полученное отображение будет недоступно

через канал поточно-ориентированного отображения и потому безопасно может использоваться в единственной go-подпрограмме (или в нескольких подпрограммах, если обеспечить очередность операций с помощью мьютекса). С этого момента доступ к отображению имеет только главная go-подпрограмма, поэтому разграничивать доступ к нему не требуется. Здесь просто выполняются итерации по парам *ключ/значение* отображения и осуществляется их вывод в консоль.

Использование значения, реализующего интерфейс `SafeMap`, обеспечивает безопасность, простоту синтаксиса и устраняет необходимость применения блокировок. Один из недостатков такого подхода – в том, что значения в поточно-ориентированном отображении имеют тип обобщенного интерфейса `interface{}`, а не конкретный тип, чем обусловлена необходимость использовать операцию приведения типа в функции `incrementer()`. (Ниже, в §7.2.4.3, будет сказано еще об одном недостатке.)

7.2.4.2. Синхронизация с помощью мьютекса

Теперь сравним и сопоставим простой и понятный подход на основе каналов с традиционным подходом на основе мьютексов. Для этого коротко рассмотрим программу `apachereport2` (в файле `apachereport2/apachereport.go`). Она представляет собой версию программы `apachereport1`, использующей пользовательский тип данных, инкапсулирующий отображение и мьютекс. Эти две программы выполняют одну и ту же работу, но по-разному. Программа `apachereport2` хранит в своем отображении целочисленные значения, а не значения типа `interface{}`, как поточно-ориентированное отображение, реализующее интерфейс `SafeMap`. Кроме того, упомянутый пользовательский тип предоставляет минимальный объем функциональных возможностей, только те, что необходимы для выполнения работы (метод `Increment()`), в противовес поточно-ориентированному отображению, реализующему законченный набор методов.

```
type pageMap struct {  
    countForPage map[string]int  
    mutex        *sync.RWMutex  
}
```

Одно из преимуществ использования конкретного типа данных заключается в отсутствии необходимости использовать обобщенный тип `interface{}`.

```
func NewPageMap() *pageMap {  
    return &pageMap{make(map[string]int), new(sync.RWMutex)}  
}
```

Эта функция возвращает готовое к использованию значение типа `*pageMap`. (В данном случае можно было бы создать указатель на мьютекс, используя выражение `&sync.RWMutex{}` вместо `new(sync.RWMutex)`; эквивалентность подобных выражений обсуждалась в §4.1.)

```
func (pm *pageMap) Increment(page string) {  
    pm.mutex.Lock()  
    defer pm.mutex.Unlock()  
    pm.countForPage[page]++  
}
```

Каждый метод, изменяющий отображение `countForPage`, должен разграничивать доступ к нему с помощью мьютекса. Здесь используется канонический шаблон: запретить мьютекс, отложить его отпирание до завершения функции (отпирание гарантируется даже в случае аварийной ситуации) и выполнить операцию доступа – в идеале выполняющуюся как можно более короткий промежуток времени.

Благодаря механизму автоматической инициализации переменных нулевыми значениями при первом обращении к странице в отображении `countForPage` (то есть когда она еще не включена в отображение) она будет добавлена в отображение со значением 0, которое немедленно будет увеличено на единицу. Соответственно, при последующих обращениях к странице, уже имеющейся в отображении, увеличиваться будет ранее сохраненное значение.

Каждый метод, обращающийся к отображению `countForPage`, должен разграничивать доступ к нему с помощью мьютекса. Для обновления значений в отображении необходимо использовать методы `sync.RWMutex.Lock()` и `sync.RWMutex.Unlock()`, как показано выше, но при обращении только для чтения можно использовать другие методы.

```
func (pm *pageMap) Len() int {  
    pm.mutex.RLock()  
    defer pm.mutex.RUnlock()  
    return len(pm.countForPage)  
}
```

Этот метод включен в пример, только чтобы показать, как использовать блокировки для чтения. Здесь используется тот же шаблон,

но блокировки для чтения потенциально более эффективны (потому что, используя их, мы сообщаем, что собираемся только читать защищенный ресурс, но не изменять его). Например, если имеются несколько go-подпрограмм, читающих значения из одного и того же отображения `countForPage`, они безопасно могут выполнять операции чтения параллельно, используя блокировку для чтения. Но если хотя бы одна из них приобретет обычную (для чтения и записи) блокировку и, следовательно, способность изменять содержимое отображения, никакая другая блокировка не сможет быть приобретена на это время.

```
pageMap.Increment(matches[1])
```

После создания отображения типа `pageMap` рабочие go-подпрограммы смогут обновлять его с помощью этой инструкции.

7.2.4.3. Синхронизация объединением локальных отображений через каналы

Используя поточно-ориентированное отображение или простое отображение, защищенное мьютексом, оправданно было бы ожидать, что увеличение количества рабочих go-подпрограмм приведет к увеличению общей скорости работы программы. Однако, поскольку операции доступа к отображениям выполняются поочередно (неявно, при использовании поточно-ориентированного отображения), с увеличением количества go-подпрограмм будет нарастать и их конкуренция за обладание ресурсом.

Как это часто бывает, добиться увеличения скорости можно за счет увеличения потребления памяти. Например, в каждой go-подпрограмме можно было бы создать собственное простое отображение. Это позволило бы увеличить пропускную способность благодаря полному отсутствию конкуренции, но ценой увеличения потребления памяти (поскольку во всех отображениях наверняка будет храниться информация об одних и тех же страницах). В конце эти отображения потребовалось бы объединить, и это стало бы узким местом в программе, потому что, пока выполняется объединение одного отображения, все остальные вынуждены будут ждать своей очереди.

Программа `apachereport3` (в файле `apachereport3/apachereport.go`) использует локальные отображения и в конце объединяет их в общее отображение. Эта программа практически идентична

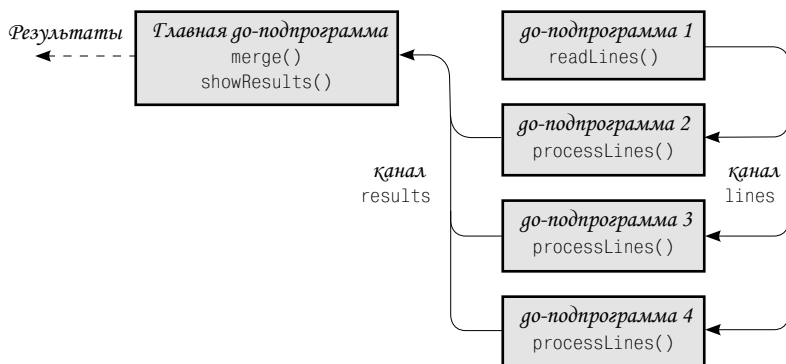


Рис. 7.7. Множество взаимозависимых заданий с синхронизацией результатов

программам `apachereport1` и `apachereport2`, поэтому далее будут рассматриваться лишь ключевые отличия. Структура параллельной обработки в программе изображена на рис. 7.7.

```
// ...
lines := make(chan string, workers*4)
results := make(chan map[string]int, workers)
go readLines(os.Args[1], lines)
getRx := regexp.MustCompile(`GET[ \t]+([^\t\n]+[.].html?)`)
for i := 0; i < workers; i++ {
    go processLines(results, getRx, lines)
}
totalForPage := make(map[string]int)
merge(results, totalForPage)
showResults(totalForPage)
// ...
```

Это фрагмент из функции `main()` программы `apachereport3`. Вместо канала `done` здесь используется канал `results`, посредством которого рабочие `go`-подпрограммы посылают локальные отображения после их заполнения. Здесь также создается общее отображение (`totalForPage`), куда объединяются все результаты.

```
func processLines(results chan<- map[string]int, getRx *regexp.Regexp,
    lines <-chan string) {
    countForPage := make(map[string]int)
    for line := range lines {
```

```

    if matches := getRx.FindStringSubmatch(line); matches != nil {
        countForPage[matches[1]]++
    }
}
results <- countForPage
}

```

Эта функция практически идентична предыдущей версии. Она имеет следующие ключевые отличия: во-первых, для хранения счетчиков обращений к страницам в каждой go-подпрограмме создается локальное отображение, и, во-вторых, по завершении обработки строк вместо отправки пустой структуры `struct{}{}` в канал `done` функция посылает локальное отображение в канал `results`.

```

func merge(results <-chan map[string]int, totalForPage map[string]int) {
    for i := 0; i < workers; i++ {
        countForPage := <-results
        for page, count := range countForPage {
            totalForPage[page] += count
        }
    }
}

```

По своей структуре эта функция идентична функции `waitUntil()`, представленной выше, но на этот раз принятое значение используется для обновления содержимого общего отображения `totalForPage`. Обратите внимание, что после отправки отображений go-подпрограммы больше не обращаются к ним, благодаря чему отпадает необходимость использования блокировок.

Функция `showResults()` почти не изменилась (поэтому она здесь не показана). Единственное ее отличие состоит в том, что она принимает отображение `totalForPage` в виде аргумента и выполняет итерации по этому отображению, выводя страницы и счетчики.

Реализация программы `apachereport3` выглядит проще и понятнее, по сравнению с программами `apachereport1` и `apachereport2`, и модель параллельной обработки, когда каждая go-подпрограмма заполняет собственную локальную структуру данных, с объединением их в конце, с успехом может использоваться в самых разных контекстах.

Конечно, для программистов, привыкших использовать парадигму блокировок, вполне естественно попробовать в программах на

языке Go использовать мьютексы для разграничения доступа. Однако документация к языку Go настоятельно рекомендует использовать go-подпрограммы и каналы и повторять мантру: «не взаимодействовать, разделяя память, а разделять память, взаимодействуя», да и компиляторы Go оптимизированы в первую очередь для поддержания именно этой модели параллельной обработки.

7.2.5. Пример: поиск дубликатов

В этом заключительном примере будет представлена программа, пытающаяся отыскать дубликаты файлов не на основе их имен, а на основе размеров и контрольных сумм SHA-1¹.

Рассматриваемая ниже программа называется `findduplicates` (в файле `findduplicates/findduplicates.go`). Для итераций по файлам и каталогам в указанном пути, включая подкаталоги, подподкаталоги и т. д., она использует функцию `filepath.Walk()` из стандартной библиотеки. Программа применяет переменное число go-подпрограмм в зависимости от выполняемой работы. Для каждого «большого» файла создается дополнительная go-подпрограмма, вычисляющая контрольную сумму SHA-1, тогда как для «маленьких» файлов вычисление контрольной суммы SHA-1 производится в текущей go-подпрограмме. Это означает, что заранее неизвестно, как много go-подпрограмм будет выполняться при каждом конкретном запуске программы, хотя можно и должно устанавливать верхний предел их количества.

Один из способов обслуживания переменного числа go-подпрограмм заключается в использовании канала для получения извещений о завершении работы, как это делалось в предыдущих примерах, и ведении учета количества созданных go-подпрограмм. Однако проще всего задействовать значение типа `sync.WaitGroup`, позволяющее добиться того же эффекта, но при этом бремя учета перекладывается на плечи окружения времени выполнения языка Go.

```
const maxGoroutines = 100
func main() {
    runtime.GOMAXPROCS(runtime.NumCPU()) // Использовать все доступные ядра
    if len(os.Args) == 1 || os.Args[1] == "-h" || os.Args[1] == "--help" {
```

¹ SHA-1 – защищенный алгоритм хеширования, воспроизводящий 20-байтные значения для любых фрагментов данных, таких как файлы. Идентичные файлы всегда будут иметь одинаковую контрольную сумму SHA-1, а различные файлы практически наверняка будут иметь разные контрольные суммы SHA-1.

```

    fmt.Printf("usage: %s <path>\n", filepath.Base(os.Args[0]))
    os.Exit(1)
}
infoChan := make(chan fileInfo, maxGoroutines*2)
go findDuplicates(infoChan, os.Args[1])
pathData := mergeResults(infoChan)
outputResults(pathData)
}

```

Функция `main()` извлекает путь к каталогу, откуда следует начинать работу, и руководит всем процессом. Она начинается с создания канала `fileInfo` для передачи значений (будет показан чуть ниже). Канал создается буферизованным, потому что, как показали эксперименты, это ведет к увеличению производительности.

Далее функция запускает функцию `findDuplicates()` в `go`-подпрограмме, после чего вызывает функцию `mergeResults()`, которая читает данные из канала `infoChan`, пока он не будет закрыт. После объединения полученных данных она возвращает их, и результаты выводятся в консоль.

Потоки взаимодействий в программе и `go`-подпрограммы изображены на рис. 7.8. Результаты передаются через канал `infoChan` в виде значений типа `fileInfo`. Они посылаются функцией «обхода» (типа `filepath.WalkFunc`), которая передается функции `filepath.Walk()`. Сама функция `filepath.Walk()` вызывается в функции `findDuplicates()`. Результаты принимаются функцией `mergeResults()`. `Go`-подпрограммы, изображенные на рис. 7.8, создаются функцией `findDuplicates()` и функцией обхода. Кроме того, функция `filepath.Walk()` из стандартной библиотеки может создавать собственные `go`-подпрограммы (например, по одной для обработки каждого каталога), хотя принцип ее действия зависит от особенностей конкретной реализации.

```

type fileInfo struct {
    sha1 []byte
    size int64
    path string
}

```

Этот тип используется для хранения информации о каждом файле. Если два файла имеют одинаковые размеры и контрольные суммы SHA-1, они считаются дубликатами независимо от путей к ним или их имен.

```
func findDuplicates(infoChan chan fileInfo, dirname string) {
    waiter := &sync.WaitGroup{}
    filepath.Walk(dirname, makeWalkFunc(infoChan, waiter))
    waiter.Wait() // Блокируется, пока вся работа не будет выполнена
    close(infoChan)
}
```

Данная функция вызывает функцию `filepath.Walk()` для обхода дерева каталогов (начиная с `dirname`), которая для каждого файла и каталога вызывает функцию `filepath.WalkFunc`, переданную ей во втором аргументе.

Функция обхода может создавать произвольное количество `go`-подпрограмм, поэтому необходим инструмент, гарантирующий, что функция `findDuplicates()` не вернет управления, пока все они не закончат работу. В данном случае роль такого инструмента играет значение типа `sync.WaitGroup`. Каждый раз, когда создается новая `go`-подпрограмма, вызывается метод `sync.WaitGroup.Add()`, а каждый раз, когда очередная `go`-подпрограмма завершает работу, вызывается метод `sync.WaitGroup.Done()`. После запуска всех `go`-подпрограмм, для ожидания их завершения, вызывается метод `sync.WaitGroup.Wait()`, который блокируется, пока количество запущенных `go`-подпрограмм не сравняется с количеством завершившихся `go`-подпрограмм.

Когда рабочие `go`-подпрограммы закончат работу, не останется ни одной, которая посылала бы значения `fileInfo` в канал `infoChan`,

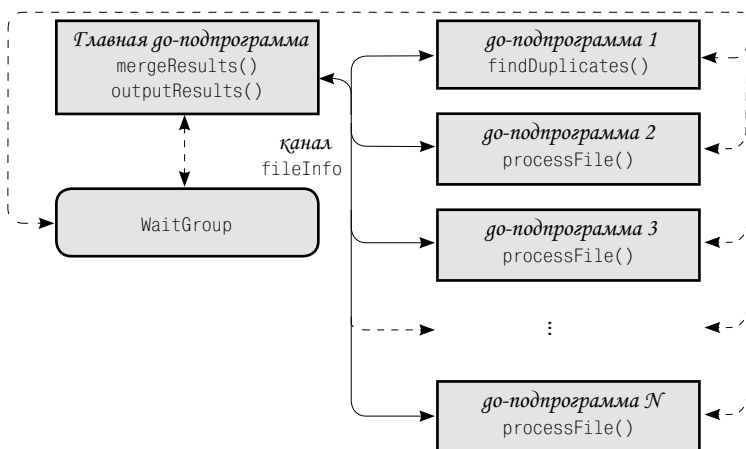


Рис. 7.8. Множество независимых заданий с синхронизацией результатов

поэтому канал закрывается. Естественно, функция `mergeResults()` сохраняет возможность принимать значения из канала, пока канал не опустеет.

```
const maxSizeOfSmallFile = 1024 * 32
func makeWalkFunc(infoChan chan fileInfo,
    waiter *sync.WaitGroup) func(string, os.FileInfo, error) error {
    return func(path string, info os.FileInfo, err error) error {
        if err == nil && info.Size() > 0 &&
            (info.Mode()&os.ModeType == 0) {
            if info.Size() < maxSizeOfSmallFile ||
                runtime.NumGoroutine() > maxGoroutines {
                processFile(path, info, infoChan, nil)
            } else {
                waiter.Add(1)
                go processFile(path, info, infoChan,
                    func() { waiter.Done() })
            }
        }
        return nil // Игнорировать любые ошибки
    }
}
```

Эта функция создает и возвращает анонимную функцию типа `filepath.WalkFunc` (то есть функцию с сигнатурой `func(string, os.FileInfo, error) error`). Эта функция будет вызываться для каждого файла и каталога, которые встретятся функции `filepath.Walk()`. Аргумент `path` — это имя файла или каталога, `info` — результаты системного вызова `stat` для этого файла или каталога и `err` — либо `nil`, либо значение с информацией об ошибке, встретившейся при попытке проанализировать имя `path`. Пропускать каталоги можно, возвращая значение ошибки `filepath.SkipDir`, а остановить обход дерева каталогов — вернув любое другое, непустое значение ошибки.

В программе решено было обрабатывать только обычные файлы ненулевого размера. (Разумеется, все файлы, имеющие размер 0 байт, являются идентичными, но программа игнорирует такие файлы.) Значение `os.ModeType` — это битовая маска, имеющая биты, обозначающие каталоги, символические ссылки, именованные каналы, сокеты и устройства, то есть если не установлен ни один из этих битов, значит, программа имеет дело с обычным файлом.

Если файл «маленький» (в данном случае до 32 Кбайт), контрольная сумма SHA-1 вычисляется немедленно, вызовом функции

`processFile()`. Но для всех остальных файлов создается новая го-подпрограмма, в которой функция `processFile()` выполняется асинхронно. Это означает, что маленькие файлы будут блокировать работу функции (пока не будет вычислена контрольная сумма SHA-1), а большие – нет, потому что для них вычисление контрольных сумм будет выполняться в отдельной го-подпрограмме. В любом случае, по завершении вычислений полученное значение `fileInfo` посылается в канал `infoChan`.

Создавая новую го-подпрограмму, необходимо вызвать метод `sync.WaitGroup.Add()`, а когда она завершает обработку, следует обязательно выполнить соответствующий вызов метода `sync.WaitGroup.Done()`. Для этой цели используется поддержка *замыканий* в языке Go (§5.6.3). Если функция `processFile()` вызывается в новой го-подпрограмме, ей в последнем аргументе передается анонимная функция, которая вызывает метод `sync.WaitGroup.Done()`. Предполагается, что функция `processFile()` выполнит отложенный вызов этой анонимной функции, чтобы обеспечить вызов метода `Done()` по завершении работы го-подпрограммы. Если функция `processFile()` вызывается в текущей го-подпрограмме, вместо анонимной функции ей в последнем аргументе передается значение `nil`.

Почему бы просто не обрабатывать *все* файлы в новых го-подпрограммах? В языке Go с этим нет никаких проблем, поскольку он позволяет создавать сотни и тысячи го-подпрограмм. К сожалению, большинство операционных систем устанавливают верхний предел на количество файлов, которые могут быть открыты одновременно. В Windows предел по умолчанию составляет 512 файлов, хотя его можно повысить до 2048. В системах Mac OS X предел по умолчанию составляет 256 файлов, в Linux – 1024, однако в Unix-подобных системах, как эти, обычно имеется возможность повысить предел до десятков и сотен тысяч. Очевидно, если создавать по одной го-подпрограмме на каждый файл, на некоторых платформах программа легко может превысить установленный предел.

Чтобы избежать проблемы «слишком большого числа открытых файлов», в программе используется комбинация из двух тактик. Во-первых, маленькие файлы обрабатываются в одной го-подпрограмме (или, возможно, в нескольких го-подпрограммах, если функция `filepath.Walk()` будет распределять работу по нескольким го-подпрограммам и вызывать функцию обхода из них). Это гарантирует, что при попадании в каталог с тысячами маленьких файлов программа не сможет открыть их слишком много, потому что только одна или несколько го-подпрограмм смогут обрабатывать их одновременно.

Большие файлы можно позволить обрабатывать в отдельных го-подпрограммах, потому что сам факт, что они большие, означает низкую вероятность, что одновременно будет открыто слишком много файлов. Но если программа встретит слишком большое количество больших файлов, то есть если она создаст слишком много го-подпрограмм, в действие вступит вторая тактика. (Функция `runtime.NumGoroutine()` позволяет узнать количество го-подпрограмм, выполняющихся в текущий момент времени.)

Если окажется, что было запущено слишком много го-подпрограмм, создание новых го-подпрограмм для больших файлов прекратится, и все файлы, независимо от размера, будут обрабатываться в текущей го-подпрограмме. Это вынудит программу использовать ту же самую го-подпрограмму (или несколько го-подпрограмм) для обработки каждого последующего файла, остановит рост количества го-подпрограмм и как следствие ограничит количество файлов, открытых одновременно. По мере завершения го-подпрограмм, обрабатывающих большие файлы, их количество будет уменьшаться. Поэтому в некоторый момент количество го-подпрограмм окажется меньше установленного предела, и тогда вновь для обработки больших файлов начнут создаваться отдельные го-подпрограммы.

```
func processFile(filename string, info os.FileInfo,
    infoChan chan fileInfo, done func()) {
    if done != nil {
        defer done()
    }
    file, err := os.Open(filename)
    if err != nil {
        log.Println("error:", err)
        return
    }
    defer file.Close()
    hash := sha1.New()
    if size, err := io.Copy(hash, file);
        size != info.Size() || err != nil {
        if err != nil {
            log.Println("error:", err)
        } else {
            log.Println("error: failed to read the whole file:", filename)
        }
        return
    }
    infoChan <- fileInfo{hash.Sum(nil), info.Size(), filename}
}
```

Эта функция вызывается из текущей go-подпрограммы или из вновь созданной дополнительной go-подпрограммы для вычисления контрольной суммы SHA-1 указанного файла и отправки информации о файле в канал `infoChan`.

Если аргумент `done` имеет значение, отличное от `nil`, следовательно, данная функция была вызвана в новой go-подпрограмме, поэтому она производит отложенный вызов функции `done()` (которая просто вызывает метод `sync.WaitGroup.Done()`). Это гарантирует, что для каждого вызова метода `sync.WaitGroup.Add()` будет выполнен парный ему вызов метода `Done()`, так как это очень важно для корректной работы функции `sync.WaitGroup.Wait()`. Если аргумент имеет значение `nil`, он просто игнорируется.

Затем указанный файл открывается для чтения, а его закрытие откладывается до завершения функции, как обычно. В пакете `crypto/sha1` из стандартной библиотеки имеется функция `sha1.New()`, возвращающая значение, реализующее интерфейс `hash.Hash`. Этот интерфейс определяет метод `Sum()`, возвращающий значение контрольной суммы (20-байтное значение хеша SHA-1), которое реализует интерфейс `io.Writer`. (Здесь методу `Sum()` передается значение `nil`, чтобы создать новый срез типа `[]byte`, однако при желании ему можно было бы передавать имеющийся срез типа `[]byte`, в конец которого была бы добавлена контрольная сумма.)

Здесь можно было бы прочитать содержимое файла целиком и затем вызвать метод `sha1.Write()` для записи файла в хеш, но в программе решено было использовать более эффективный подход и применить функцию `io.Copy()`. Эта функция принимает значение, реализующее запись (в данном случае – хеш), и значение, реализующее чтение (здесь – открытый файл), и копирует из последнего в первое. Когда копирование завершится, `io.Copy()` вернет количество скопированных байт и либо значение `nil`, либо значение типа `error` в случае ошибки. Поскольку хеш SHA-1 может работать сразу с целыми фрагментами данных, максимальный объем памяти, используемой функцией `io.Copy()`, будет равен размеру буфера, используемого хешем SHA-1, плюс некоторый фиксированный объем. Если читать файлы в память целиком, расходовался бы тот же объем памяти, плюс память для хранения всего файла. Поэтому, особенно для больших файлов, функция `io.Copy()` позволяет получить ощутимую экономию.

Когда вычисления завершатся, в канал `infoChan` посылается значение типа `FileInfo` с полученной контрольной суммой SHA-

размером файла (доступен в значении типа `os.FileInfo`, передаваемом функции `processFile()` из функции обхода) и именем файла (включая полный путь).

```
type pathsInfo struct {
    size int64
    paths []string
}
```

Значения этого типа используются для хранения информации о каждом повторяющемся файле, то есть его размер, а также все пути и имена файлов. Значения данного типа используются функциями `mergeResults()` и `outputResults()`.

```
func mergeResults(infoChan <-chan fileInfo) map[string]*pathsInfo {
    pathData := make(map[string]*pathsInfo)
    format := fmt.Sprintf("%%016X:%%dX", sha1.Size*2) // == "%016X:%40X"
    for info := range infoChan {
        key := fmt.Sprintf(format, info.size, info.sha1)
        value, found := pathData[key]
        if !found {
            value = &pathsInfo{size: info.size}
            pathData[key] = value
        }
        value.paths = append(value.paths, info.path)
    }
    return pathData
}
```

Эта функция начинается с создания отображения для сохранения информации о дубликатах файлов. Ключами в нем являются строки, составленные из размера файла, двоеточия и контрольной суммы SHA-1, а значениями — указатели `*pathsInfo`.

Строка `format` используется для создания ключей, состоящих из 16 шестнадцатеричных цифр с ведущими нулями, представляющих размер файла, и некоторого количества шестнадцатеричных цифр, достаточного для представления контрольной суммы SHA-1 файла. Ведущие нули в части, представляющей размер файла, используются с целью упростить сортировку по ключам. Константа `sha1.Size` хранит количество байт в контрольной сумме SHA-1 (то есть 20). Так как один байт представлен двумя шестнадцатеричными цифрами, число, определяющее в строке формата ширину поля для вывода

контрольной суммы SHA-1, должно быть вдвое больше количества байт. (Между прочим, строку формата можно было бы создать немного иначе: `format = "%016X:%" + fmt.Sprintf("%dX", sha1.Size*2).`)

Посылка данных в канал `infoChan` выполняется множеством го-подпрограмм, но прием данных канала производит только одна функция (в главной го-подпрограмме). Цикл `for` принимает значения типа `fileInfo` или блокируется в их ожидании. Когда из канала `infoChan` будут приняты все значения и канал закроется, цикл завершится. Для каждого принятого значения типа `fileInfo` создается строка с ключом в отображении. Если элемент с таким ключом отсутствует в отображении, создается соответствующее значение с указанным размером файла и пустым срезом, которое добавляется в отображение с новым ключом. Затем в конец поля `paths` нового или существовавшего ранее элемента отображения добавляется путь к файлу из значения типа `fileInfo`.

В конце это приведет к тому, что для повторяющихся файлов срезы будут хранить более одного пути, а для неповторяющихся – только один. После обработки всех значений типа `fileInfo` и заполнения отображения функция вернет отображение, готовое к дальнейшей обработке.

```
func outputResults(pathData map[string]*pathsInfo) {
    keys := make([]string, 0, len(pathData))
    for key := range pathData {
        keys = append(keys, key)
    }
    sort.Strings(keys)
    for _, key := range keys {
        value := pathData[key]
        if len(value.paths) > 1 {
            fmt.Printf("%d duplicate files (%s bytes):\n",
                len(value.paths), commas(value.size))
            sort.Strings(value.paths)
            for _, name := range value.paths {
                fmt.Printf("\t%s\n", name)
            }
        }
    }
}
```

Ключи отображения `pathData` являются строками, начинающимися с 16-значных шестнадцатеричных значений размеров файлов

с ведущими нулями. (16 цифр было выбрано потому, что этого их количества достаточно для представления любых чисел типа `int64`.) Это означает, что, выполняя сортировку по ключам, можно отсортировать элементы отображения по размеру файлов. Поэтому функция начинается с создания среза `keys`, после чего в него добавляются ключи из отображения и выполняется сортировка среза. Затем функция выполняет итерации по отсортированному срезу с ключами и извлекает соответствующие им значения типа `pathsInfo`. Для значений, где имеются несколько путей, выводится размер файла и затем с отступом список путей к повторяющимся файлам в алфавитном порядке, как показано ниже.

```
$ ./findduplicates $GOROOT
2 duplicate files (67 bytes):
    /home/mark/opt/go/test/fixedbugs/bug248.dir/bug0.go
    /home/mark/opt/go/test/fixedbugs/bug248.dir/bug1.go
...
4 duplicate files (785 bytes):
    /home/mark/opt/go/doc/gopher/gophercolor16x16.png
    /home/mark/opt/go/favicon.ico
    /home/mark/opt/go/misc/dashboard/godashboard/static/favicon.ico
    /home/mark/opt/go/src/pkg/archive/zip/testdata/gophercolor16x16.png
...
2 duplicate files (1,371,249 bytes):
    /home/mark/opt/go/bin/ebnflint
    /home/mark/opt/go/src/cmd/ebnflint/ebnflint
```

Здесь было опущено множество строк, обозначенных многоточиями.

```
func commas(x int64) string {
    value := fmt.Sprintf(x)
    for i := len(value) - 3; i > 0; i -= 3 {
        value = value[:i] + "," + value[i:]
    }
    return value
}
```

Большинству пользователей сложно воспринимать на глаз длинные числа (например, 1371249), поэтому здесь использована простая функция `commas()`, вставляющая запятые между группами разрядов, чтобы упростить восприятие значений размеров файлов. Функция принимает единственное значение типа `int64`, то есть если имеется значение типа `int` или `integer` другого целочисленного типа,

его необходимо преобразовать в значение типа `int64`, например: `commas(int64(i))`¹.

На этом завершается наш обзор программы `findduplicates` и приемов параллельного программирования на языке Go в целом. Поддержка параллельного выполнения в языке Go (`<-`, `chan`, `go`, `select`) обладает большой гибкостью и допускает такое количество способов организации параллельной обработки данных, что здесь не хватило бы места, чтобы показать их все. Тем не менее примеры и упражнения, следующие ниже, дают достаточный объем знаний и навыков параллельного программирования на языке Go, чтобы уверенно применять их при создании новых программ параллельной обработки.

Конечно, невозможно сказать определенно, какой из рассмотренных (или других возможных) подходов является лучшим, так как каждый из них может оказаться лучшим при различных обстоятельствах. Производительность каждого из подходов во многом зависит от конкретного компьютера, количества `go`-подпрограмм и от местоположения обрабатываемых данных – в памяти или на внешних носителях (например, в сети). Надежный способ выбора лучшего подхода для конкретной программы заключается в том, чтобы провести тестирование на фактических данных с использованием разных подходов и с разным числом `go`-подпрограмм.

7.3. Упражнения

В этой главе предлагается выполнить три упражнения. Первое связано с созданием поточно-ориентированной структуры данных. Во втором и третьем потребуется создать небольшие, но законченные программы, при этом над третьим упражнением придется хорошо подумать.

1. Создайте тип поточно-ориентированного среза с именем `safeSlice`, реализующий следующий экспортируемый интерфейс `SafeSlice`:

```
type SafeSlice interface {
    Append(interface{}) // Добавляет элемент в конец среза
    At(int) interface{} // Возвращает элемент с указанным индексом
    Close() []interface{} // Закрывает канал и возвращает срез
    Delete(int)           // Удаляет элемент с указанным индексом
    Len() int             // Возвращает количество элементов в срезе
    Update(int, UpdateFunc) // Обновляет элемент с указанным индексом
}
```

¹ На момент написания этих строк в языке Go отсутствовала поддержка региональных настроек.

Здесь также потребуется создать метод `safeSlice.run()`, в котором будет создаваться фактический срез (типа `[]interface{}`), выполняющий «бесконечный» цикл по элементам канала, и функцию `New()`, создающую поточно-ориентированный срез и запускающую метод `safeSlice.run()` в го-подпрограмме.

Поточно-ориентированный срез легко можно создать по аналогии с поточно-ориентированным отображением, описанным выше (§7.2.3), с методом `safeSlice.Update()`, точно так же подверженным опасности взаимоблокировки. Одно из возможных решений находится в файле `safeslice/safeslice.go` и занимает около 100 строк. (Для тестирования среза можно использовать программу `apachereport4`, использующую пакет `safeslice`.)

2. Создайте программу, принимающую одно или более имен файлов изображений в командной строке, которая для каждого из них будет выводить в консоль HTML-тег в форме: ``. Программа должна обрабатывать имена файлов изображений параллельно, используя фиксированное число рабочих го-подпрограмм. Порядок вывода не имеет значения (при условии, что каждая строка с тегом будет выводиться полностью!). Имена файлов должны выводиться без путей. В командной строке допускается указывать любое количество файлов, но при этом имена, не являющиеся именами обычных файлов или именами файлов изображений, должны игнорироваться, аналогично должны игнорироваться любые ошибки.

В стандартной библиотеке имеется функция `image.DecodeConfig()`, способная извлекать ширину и высоту изображения из значения, реализующего интерфейс `io.Reader` (возвращаемого функцией `os.Open()`), не читая файла изображения целиком. Чтобы эта функция могла распознавать различные форматы изображений (`.jpg`, `.png` и др.), необходимо импортировать соответствующие пакеты. Однако здесь не потребуется использовать эти пакеты непосредственно, поэтому, чтобы избежать появления сообщений компилятора, таких как «imported and not used» (импортирован, но не используется), пакеты должны импортироваться в пустой идентификатор (например, `_ "image/jpeg"`, `_ "image/png"` и т. д.). Эти способы импортирования будут обсуждаться в следующей главе.

В загружаемые примеры к книге включены два возможных решения: `imagetag1` — однопоточное (без дополнительных

go-подпрограмм и каналов) и `imagetag2` — многопоточное, похожее на реализацию примера `cgrep2`. Желаящие сосредоточиться на аспектах параллельной обработки могут просто скопировать файл `imagetag1/imagetag1.go` в каталог, например `my_imagetag`, и превратить эту однопоточную реализацию в многопоточную. Для этого придется изменить функции `main()` и `process()` и добавить примерно 40 строк программного кода. Пользователи Windows могут воспользоваться функцией `commandLineFiles()` (§4.4.2) для обработки шаблонных символов в именах файлов. Оба готовых решения уже включают ее. Наиболее уверенные в своих силах читатели могут попробовать написать программу с самого начала. В файле `imagetag2/imagetag2.go` приводится решение, использующее один из возможных подходов и занимающее около 100 строк.

3. Создайте программу с поддержкой параллельной обработки, использующую фиксированное число рабочих go-подпрограмм, которая будет принимать имена одного или более HTML-файлов в командной строке. Программа должна читать содержимое HTML-файлов, в каждом встреченном теге `` проверять наличие атрибутов `width` и `height` и в случае их отсутствия определять ширину и высоту указанного изображения и добавлять отсутствующие атрибуты. Параллельная обработка может быть реализована по аналогии с примерами `apachereport` (здесь не имеется в виду версия `apachereport2` на основе мьютексов), за исключением канала передачи результатов, который тут не нужен.

Поскольку в версии 1 в стандартной библиотеке Go отсутствует парсер разметки HTML, для реализации данного упражнения предлагается использовать пакеты `regexp` и `strings`¹. Поиск тегов изображений предлагается выполнять с помощью регулярного выражения ``<[iI][mM][gG][^>]+>``, а поиск имен файлов внутри тегов — с помощью регулярного выражения ``src=["']([~"']+)["']``. (Это не слишком сложные регулярные выражения, потому что основное внимание здесь уделяется организации параллельной обработки, а не регулярным выражениям, которые описываются в §3.6.5.) Для определения размеров изображения можно использовать функцию `image`.

¹ Парсер разметки HTML находится в разработке и может быть доступен к моменту, когда вы будете читать эти строки.

`DecodeConfig()`, упоминавшуюся в предыдущем упражнении (вместе с импортируемыми пакетами). Как обычно, пользователи Windows могут использовать функцию `commandLineFiles()` (§4.4.2) для обработки шаблонных символов в именах файлов. Организация параллельной обработки данных в этой программе достаточно проста, но сама обработка сопряжена с определенными сложностями. Это является выгодным отличием от некоторых других языков программирования, где сосредоточенность на аспектах параллельной обработки существенно мешает реализации фактической обработки.

В загружаемые примеры к книге включены два возможных решения. Первое решение, `sizeimages1` (в файле `sizeimages1/sizeimages1.go`), делает все, что необходимо, но имеет один недостаток: оно способно находить файлы изображений, только если они находятся в тех же каталогах, что и HTML-файлы. Это ограничение обусловлено тем, что каждый тег `` замещается измененной версией тега вызовом метода `regex.Regexp.ReplaceAllStringFunc()`. Этот метод ожидает получить функцию, выполняющую замену, с сигнатурой `func(string) string`, которой в качестве аргумента передается строка с совпавшим текстом, а она должна вернуть замещающую строку. В качестве примера типичного аргумента, передаваемого этой функции, можно привести строку ``. Функция замены понятия не имеет, где искать файл `splash.png`, и поэтому предполагает, что он находится в текущем каталоге. Как следствие этого ограничения программа `sizeimages1` должна запускаться в том же каталоге, где находятся HTML-файлы.

Можно было бы попробовать решить эту проблему, определив глобальную переменную `directory` и перед вызовом функции замены добавляя к ее значению путь к текущему HTML-файлу, но такой прием годится не всегда. Почему? Второй способ, `sizeimages2` (в файле `sizeimages2/sizeimages2.go`), решает описанную проблему созданием новой функции замены в виде замыкания, сохраняющей путь к каталогу с HTML-файлом каждый раз, когда это необходимо. Затем функция замены использует сохраненный путь к каталогу для определения полного пути к файлам изображений, в которых указан относительный путь.

Это, пожалуй, самое сложное упражнение из встречавшихся до сих пор в данной книге, требующее применения функций из пакетов `image`, `regex` и `strings`. Файл `sizeimages1.go` содержит около 160 строк, а файл `sizeimages2.go` – около 170.



8. Обработка файлов

В предыдущих главах было показано несколько примеров операций создания, чтения и записи текстовых файлов. В этой главе мы более подробно рассмотрим имеющиеся в языке Go инструменты для работы с файлами и в особенности способы чтения и записи файлов стандартных форматов (таких как XML и JSON), а также пользовательских текстовых и двоичных форматов.

К настоящему моменту были охвачены все особенности языка Go (кроме создания программ с собственными и сторонними пакетами, о которых рассказывается в следующей главе), поэтому теперь можно свободно использовать любые возможности, предоставляемые языком Go. Теперь мы можем, пользуясь этой свободой, задействовать замыкания (§5.6.3), чтобы избежать дублирования программного кода, и в некоторых случаях более широко использовать поддержку объектно-ориентированного программирования, создавая методы добавок к функциям.

В этой главе основное внимание будет уделено не каталогам или файловым системам, а файлам. Что касается каталогов, пример `findduplicates` из предыдущей главы (§7.2.5) демонстрирует, как выполнять итерации по файлам и подкаталогам с помощью функции `filepath.Walk()`. Кроме того, тип `os.File` из пакета `os` в стандартной библиотеке реализует методы чтения имен в каталоге (`os.File.Readdirnames()`) и извлечения значений типа `os.FileInfo` для каждого элемента в каталоге (`os.File.Readdir()`).

В первом разделе этой главы рассказывается, как писать и читать файлы стандартных и пользовательских форматов. Второй раздел охватывает поддержку обработки файлов-архивов и сжатых файлов.

8.1. Файлы с пользовательскими данными

Для программ весьма характерно хранить данные во внутреннем представлении и предоставлять возможность импорта/экспорта

для поддержки обмена данными, а также чтобы упростить обработку данных внешними инструментами. Поскольку здесь речь идет об обработке файлов, основное внимание будет сосредоточено исключительно на том, как писать и читать данные в стандартных и пользовательских форматах в/из внутреннего их представления в программе.

Во всех примерах в этом разделе будут использоваться одни и те же данные, чтобы получить возможность сравнить различные форматы файлов. Весь программный код взят из программы `invoicedata` (файлы `invoicedata.go`, `gob.go`, `inv.go`, `jsn.go`, `txt.go` и `xml.go` в каталоге `invoicedata`). Эта программа принимает в виде аргументов командной строки имена двух файлов, один для чтения и один для записи (то есть это должны быть разные имена файлов). Затем она читает данные из первого файла (формат представления которых определяется по расширению в имени файла) и записывает их во второй файл (опять же, в формате, определяемом по расширению в имени файла).

Файлы, созданные программой `invoicedata`, являются кросс-платформенными, то есть файл, созданный в Windows, сможет быть прочитан в Mac OS X и Linux, и наоборот, независимо от формата. Сжатые `gzip`-файлы (например, `invoices.gob.gz`) могут читаться и записываться программой без применения дополнительных инструментов (вопросы сжатия файлов рассматриваются во втором разделе (§8.2)).

Данные в программе хранятся в виде значения типа `[]*Invoice`, то есть в виде среза с *указателями*¹ на значения типа `Invoice`. Информация о каждой накладной¹ хранится в виде значения типа `Invoice`, и каждая накладная содержит нуль или более пунктов в своем поле `Items` типа `[]*Item` (срез с указателями на значения типа `Item`).

type <code>Invoice</code> struct {	type <code>Item</code> struct {
<code>Id</code> <code>int</code>	<code>Id</code> <code>string</code>
<code>CustomerId</code> <code>int</code>	<code>Price</code> <code>float64</code>
<code>Raised</code> <code>time.Time</code>	<code>Quantity</code> <code>int</code>
<code>Due</code> <code>time.Time</code>	<code>Note</code> <code>string</code>
<code>Paid</code> <code>bool</code>	}
<code>Note</code> <code>string</code>	
<code>Items</code> <code>[]*Item</code>	
}	

¹ Имя типа `Invoice` переводится как «накладная». — *Прим. перев.*

Эти две структуры используются для хранения данных. В табл. 8.1 показаны некоторые результаты измерения времени чтения и записи одних и тех же 50 000 накладных, и размеры файлов каждого поддерживаемого формата. Результаты измерения времени приводятся в секундах, с округлением до ближайшей десятой доли секунды, их не следует рассматривать как абсолютные значения, потому что на разных компьютерах, несомненно, будут получены разные результаты. В столбце «Размер» приводятся размеры файлов в килобайтах – эти значения не должны изменяться на разных компьютерах. Для данного набора данных размеры сжатых файлов удивительно близки друг к другу, даже при том, что размеры несжатых файлов значительно отличаются. В столбце, где приводится количество строк программного кода, не учитывается код, общий для всех форматов (например, код сжатия и распаковывания, а также объявления структур).

Таблица 8.1. Сравнительные характеристики скорости обработки и размеров различных форматов

Расширение	Чтение	Запись	Размер (Кбайт)	Чтение/запись количество строк	Формат
.gob	0.3	0.2	7948	21 + 11 = 32	Двоичный Go
.gob.gz	0.5	1.5	2589		
.jsn	4.5	2.2	16283	32 + 17 = 49	JSON
.jsn.gz	4.5	3.4	2678		
.xml	6.7	1.2	18917	45 + 30 = 75	XML
.xml.gz	6.9	2.7	2730		
.txt	1.9	1.0	12375	86 + 53 = 139	Простой текст (UTF-8)
.txt.gz	2.2	2.2	2514		
.inv	1.7	3.5	7250	128 + 87 = 215	Пользовательский двоичный
.inv.gz	1.6	2.6	2400		

Результаты хронометража и размеры файлов вполне ожидаемы, кроме разве что необычно высокой скорости чтения и записи данных в простом текстовом формате. Это обусловлено сочетанием прекрасной реализации функций ввода и вывода в пакете `fmt` и специального текстового формата, созданного с целью максимально упростить парсинг данных. В реализации поддержки форматов JSON и XML вместо формата по умолчанию для значений типа `time.Time` (строки в формате ISO-8601, включающие дату и время) сохраняется только дата. Это позволило немного уменьшить размеры файлов за

счет незначительной потери скорости обработки и дополнительного программного кода. Например, программный код, обрабатывающий формат JSON, мог бы выполняться быстрее и занимал бы примерно столько же строк, сколько занимает программный код обработки двоичного формата Go, если бы преобразования значений типа `time.Time` выполнялись самой реализацией поддержки формата JSON.

Из двоичных форматов наиболее удобным является двоичный формат Go – он очень быстрый, чрезвычайно компактный, для его поддержки требуется небольшой объем программного кода, и он относительно просто адаптируется к изменениям в данных. Однако при использовании пользовательских типов данных, которые изначально не поддерживают формат `gob`, необходимо реализовать в них методы интерфейсов `gob.Encoder` и `gob.Decoder`, это может существенно замедлить операции чтения и записи данных в формате `gob`, а также привести к раздуванию размеров файлов.

С точки зрения восприятия человека, лучшим форматом представления данных является, пожалуй, формат XML, в частности потому, что он широко распространен, как формат обмена данными. Для обработки формата XML потребовалось больше программного кода, чем для формата JSON. Это обусловлено тем, что в версии Go 1 отсутствует `xml.Marshaler` (который, как ожидается, появится в одной из более поздних версий Go 1.x), а также использованием параллельных типов для упрощения отображения данных между XML-форматом (`XMLInvoice` и `XMLItem`) и внутренним представлением (`Invoice` и `Item`). Приложения, экспортирующие свои данные в формате XML, могут не использовать параллельные типы данных или преобразования, используемые в программе `invoicedata`, поэтому они могут выполняться быстрее и занимать меньше строк программного кода, чем программа `invoicedata`.

Кроме скорости чтения и записи, размеров файлов и количества строк программного кода, существует еще одна проблема, которую необходимо учитывать: надежность формата. Например, если добавить одно поле в структуру `Invoice` и одно поле в структуру `Item`, это потребует изменить форматы файлов. Как проще адаптировать программный код для чтения и записи нового формата, сохранив при этом возможность чтения старого формата? Добавив анализ информации о версии в описание форматов файлов, подобного рода адаптация реализуется достаточно просто (как демонстрирует одно из упражнений к этой главе), исключение составляет только формат JSON, для которого адаптация программного кода для

чтения и записи старого и нового форматов реализуется немного сложнее.

В дополнение к структурам `Invoice` и `Item` в реализациях поддержки форматов файлов используются следующие константы:

```
const (
    fileType      = "INVOICES" // Используется текстовыми форматами
    magicNumber    = 0x125D     // Используется двоичными форматами
    fileVersion    = 100        // Используется всеми форматами
    dateFormat     = "2006-01-02" // Эта дата всегда должна использоваться
)
```

Константа `magicNumber` используется для идентификации файлов с накладными¹. Константа `fileVersion` определяет версию формата файлов с накладными – ее использование упростит возможность изменения программы в будущем для адаптации к изменениям в структуре данных. Константа `dateFormat` демонстрирует, как формируются даты, и будет обсуждаться ниже.

Здесь также определена пара интерфейсов.

```
type InvoicesMarshaler interface {
    MarshalInvoices(writer io.Writer, invoices []*Invoice) error
}
type InvoicesUnmarshaler interface {
    UnmarshalInvoices(reader io.Reader) ([]*Invoice, error)
}
```

Их цель – упростить чтение и запись специализированных форматов обобщенным способом. Например, ниже приводится функция из программы `invoicedata`, используемая для чтения накладных из открытого файла.

```
func readInvoices(reader io.Reader, suffix string) ([]*Invoice, error) {
    var unmarshaler InvoicesUnmarshaler
    switch suffix {
    case ".gob":
        unmarshaler = GobMarshaler{}
    case ".inv":
        unmarshaler = InvMarshaler{}
    }
```

¹ В мире не существует какого-то централизованного репозитория сигнатур файлов, поэтому нельзя с уверенностью сказать, что какая-либо сигнатура не использовалась ранее.

```

case ".json", ".json":
    unmarshaller = JSONMarshaler{}
case ".txt":
    unmarshaller = TxtMarshaler{}
case ".xml":
    unmarshaller = XMLMarshaler{}
}
if unmarshaller != nil {
    return unmarshaller.UnmarshalInvoices(reader)
}
return nil, fmt.Errorf("unrecognized input suffix: %s", suffix)
}

```

Аргумент `reader` – это значение любого типа, реализующего интерфейс `io.Reader`, такое как открытый файл (типа `*os.File`), декомпрессор `gzip` (типа `*gzip.Reader`) или значение типа `string.Reader`. Аргумент `suffix` – расширение имени файла (оставшееся после удаления расширения `.gz`). Структуры `GobMarshaler`, `InvMarshaler` и др. – это пользовательские типы, реализующие методы `MarshalInvoices()` и `UnmarshalInvoices()` (и, соответственно, реализующие интерфейсы `InvoicesMarshaler` и `InvoicesUnmarshaler`), как будет показано в следующих подразделах.

8.1.1. Обработка файлов в формате JSON

Согласно описанию на сайте www.json.org, JSON (JavaScript Object Notation – форма записи объектов JavaScript) – это легковесный формат обмена данными, простой для восприятия человеком и простой для обработки программным способом. Формат JSON – это простой текстовый формат в кодировке UTF-8. Он приобретает все большую популярность (особенно для передачи данных через сетевые соединения), потому что он более удобный для записи, более компактный (обычно) и требует меньше вычислительных ресурсов для парсинга, чем формат XML.

Ниже приводится пример записи одной накладной в формате JSON (большая часть полей второго пункта накладной опущена).

```

{
    "Id": 4461,
    "CustomerId": 917,
    "Raised": "2012-07-22",
    "Due": "2012-08-21",

```

```
"Paid": true,
"Note": "Use trade entrance",
"Items": [
    {
        "Id": "AM2574",
        "Price": 415.8,
        "Quantity": 5,
        "Note": ""
    },
    {
        "Id": "MI7296",
        ...
    }
]
}
```

Обычно инструменты, имеющиеся в пакете `encoding/json`, выводят данные в формате JSON без лишних пробелов, но здесь были добавлены дополнительные пробелы и отступы, чтобы проще было охватить ее взглядом. Несмотря на то что пакет `encoding/json` поддерживает значения типа `time.Time`, в данной программе реализованы собственные методы `MarshalJSON()` и `UnmarshalJSON()` типа `Invoice` для обработки дат. Это позволило сократить строки с датами (за счет удаления значения времени, которое в данной программе всегда равно нулю), например до `"2012-09-06"` вместо `"2012-09-06T00:00:00Z"`.

8.1.1.1. Запись файлов в формате JSON

Чтобы реализовать методы `MarshalInvoices()` и `UnmarshalInvoices()` для работы с форматом JSON, в программе был определен тип на основе пустой структуры.

```
type JSONMarshaler struct{}
```

Этот тип реализует обобщенные интерфейсы `InvoicesMarshaler` и `InvoicesUnmarshaler`, представленные выше.

Ниже приводится реализация метода записи полного набора данных типа `[]*Invoice` в значение типа `io.Writer` в формате JSON с использованием механизмов преобразования, реализованных в пакете `encoding/json`. Значение, куда выполняется запись, должно быть значением типа `*os.File`, возвращаемым функцией `os.Create()`,

или значением типа `*gzip.Writer`, возвращаемым функцией `gzip.NewWriter()`, или значением любого другого типа, реализующего интерфейс `io.Writer`.

```
func (JSONMarshaler) MarshalInvoices(writer io.Writer,
    invoices []*Invoice) error {
    encoder := json.NewEncoder(writer)
    if err := encoder.Encode(fileType); err != nil {
        return err
    }
    if err := encoder.Encode(fileVersion); err != nil {
        return err
    }
    return encoder.Encode(invoices)
}
```

Тип `JSONMarshaler` не имеет данных, поэтому нет необходимости присваивать его значение переменной-приемнику.

Метод начинается с создания значения для преобразования данных в формат JSON, которое обортывает значение, реализующее интерфейс `io.Writer`, куда можно будет записывать данные для преобразования в формат JSON.

Запись данных выполняется вызовом метода `json.Encoder.Encode()`. Этот метод отлично справляется со срезом с накладными, каждая из которых содержит свой срез с пунктами. Метод возвращает значение ошибки или `nil`. В случае ошибки она немедленно возвращается вызывающей программе.

Строго говоря, запись типа файла и версии не является необходимой, но, как иллюстрирует одно из упражнений, это упростит изменение формата файла в будущем (например, в случае добавления новых полей в структуры `Invoice` и `Item`) и обеспечит возможность чтения файлов обоих форматов, старого и нового.

Обратите внимание, что в этом методе никак не учитываются типы кодируемых данных, поэтому не составит труда написать функции для записи других данных, поддерживающих возможность преобразования в формат JSON. Кроме того, для организации записи данных в новом формате не потребуется изменять метод `JSONMarshaler.MarshalInvoices()`, при условии что все новые поля будут экспортируемыми и поддерживающими возможность преобразования в формат JSON.

Если бы в программе не предъявлялось особых требований к преобразованию данных в формат JSON, этого программного кода было бы вполне достаточно. Однако из-за необходимости более точного управления выводом данных в формате JSON – в частности, для форматирования значений типа `time.Time` – необходимо снабдить тип `Invoice` методом `MarshalJSON()`, реализующим интерфейс `json.Marshaler`. Функция `json.Encode()` автоматически проверяет поддержку интерфейса `json.Marshaler` преобразуемым значением и при ее наличии вызывает метод `MarshalJSON()` преобразуемого значения вместо встроенной процедуры преобразования.

```
type JSONInvoice struct {
    Id          int
    CustomerId int
    Raised      string // поле типа time.Time в структуре Invoice
    Due         string // поле типа time.Time в структуре Invoice
    Paid        bool
    Note        string
    Items       []*Item
}

func (invoice Invoice) MarshalJSON() ([]byte, error) {
    jsonInvoice := JSONInvoice{
        invoice.Id,
        invoice.CustomerId,
        invoice.Raised.Format(dateFormat),
        invoice.Due.Format(dateFormat),
        invoice.Paid,
        invoice.Note,
        invoice.Items,
    }
    return json.Marshal(jsonInvoice)
}
```

Метод `Invoice.MarshalJSON()` принимает значение типа `Invoice` и возвращает его версию в формате JSON. Первая инструкция в методе просто копирует поля накладной в структуру `JSONInvoice`, преобразуя два значения типа `time.Time` в строки. Поскольку все поля в структуре `JSONInvoice` являются либо логическими, либо числовыми, либо строковыми значениями, она может быть преобразована в формат JSON с помощью функции `json.Marshal()`, поэтому здесь практически всю работу выполняет эта функция.

Для записи даты/времени (то есть значений типа `time.Time`) в виде строк необходимо использовать метод `time.Time.Format()`. Этот метод принимает строку формата, определяющую, в каком виде должна быть записана дата/время. Строка формата имеет несколько необычный вид — она всегда должна быть строковым представлением значения 1136243045 времени Unix, которое точно соответствует дате и времени 2006-01-02T15:04:05Z07:00, или некоторым подмножеством этой даты и времени. Дата и время здесь выбраны произвольно, но это значение должно быть фиксированным — никакое другое значение не должно использоваться для определения форматов даты, времени и даты/времени.

Если потребуется определить другие форматы представления даты/времени, они всегда должны записываться в терминах языка Go. Например, если потребуется выводить дату в формате: день недели, месяц, число, год, — следует использовать строку формата, такую как "Mon, Jan 02, 2006" или "Mon, Jan _2, 2006", если необходимо подавить вывод ведущих нулей. Полную информацию по этой теме, а также список некоторых предопределенных строк формата можно найти в документации к пакету `time`.

8.1.1.2. Чтение файлов в формате JSON

Чтение данных в формате JSON выполняется так же просто, как и запись, особенно если чтение выполняется в переменные того же типа, что и переменные, откуда выполнялась запись. Метод `JSONMarshaler.UnmarshalInvoices()` принимает значение с поддержкой интерфейса `io.Reader`, которое может быть значением типа `*os.File`, возвращаемым функцией `os.Open()`, или `*gzip.Reader`, возвращаемым функцией `gzip.NewReader()`, или значением любого другого типа, реализующего интерфейс `io.Reader`.

```
func (JSONMarshaler) UnmarshalInvoices(reader io.Reader) ([]*Invoice,
    error) {
    decoder := json.NewDecoder(reader)
    var kind string
    if err := decoder.Decode(&kind); err != nil {
        return nil, err
    }
    if kind != fileType {
        return nil, errors.New("cannot read non-invoices json file")
    }
    var version int
```

```

if err := decoder.Decode(&version); err != nil {
    return nil, err
}
if version > fileVersion {
    return nil, fmt.Errorf("version %d is too new to read", version)
}
var invoices []*Invoice
err := decoder.Decode(&invoices)
return invoices, err
}

```

Программе требуется прочитать три элемента данных: тип файла, версию файла и собственно накладные. Метод `json.Decoder.Decode()` принимает указатель на значение, заполненное данными в формате JSON для обратного преобразования, и возвращает признак ошибки или `nil`. Чтобы убедиться, что файл действительно содержит информацию о накладных и его версия поддерживается программой, здесь используются две переменные (`kind` и `version`). Далее выполняется чтение накладных, в процессе которого метод `json.Decoder.Decode()` будет увеличивать размер среза `invoices` по мере его заполнения и сохранять в нем указатели на значения типа `Invoice` (и `Item`), создаваемые на лету по мере необходимости. В конце метод возвращает срез `invoices` и `nil` или `nil` и признак ошибки.

Если полагаться исключительно на функциональность пакета `json` и для преобразования полей `Raised` и `Due` использовать формат по умолчанию, данного метода было бы вполне достаточно. Однако из-за того, что в программе выбран особый подход к преобразованию полей `Raised` и `Due` типа `time.Time` (сохраняется только дата), необходимо реализовать метод обратного преобразования значения даты в нестандартном формате.

```

func (invoice *Invoice) UnmarshalJSON(data []byte) (err error) {
    var jsonInvoice JSONInvoice
    if err = json.Unmarshal(data, &jsonInvoice); err != nil {
        return err
    }
    var raised, due time.Time
    if raised, err = time.Parse(dateFormat, jsonInvoice.Raised);
        err != nil {
        return err
    }
    if due, err = time.Parse(dateFormat, jsonInvoice.Due); err != nil {

```



```
        return err
    }
    *invoice = Invoice{
        jsonInvoice.Id,
        jsonInvoice.CustomerId,
        raised,
        due,
        jsonInvoice.Paid,
        jsonInvoice.Note,
        jsonInvoice.Items,
    }
    return nil
}
```

Этот метод использует ту же структуру `JSONInvoice` и заполняет ее данными о накладной с помощью той же стандартной функции `json.Unmarshal()`. Затем создается значение типа `Invoice`, куда копируются данные с датами, преобразованными в значения типа `time.Time`.

Естественно, метод `json.Decoder.Decode()` также автоматически проверяет поддержку интерфейса `json.Unmarshaler` декодируемым значением и при ее наличии использует метод `UnmarshalJSON()`, определяемый значением.

В случае добавления в данные с накладными новых экспортируемых полей этот метод будет их обрабатывать точно так же, при условии что метод `Invoice.UnmarshalJSON()` будет учитывать версию файла. Кроме того, если нулевые значения для новых полей будут недопустимы, тогда после чтения файлов в первоначальном формате необходимо предусмотреть дополнительную постобработку данных, чтобы присвоить новым полям допустимые значения по умолчанию. (Одно из упражнений в конце главы как раз будет связано с добавлением новых полей и реализацией подобного рода постобработки.)

Формат JSON очень прост в обработке, особенно если создать подходящие структуры с экспортируемыми полями, однако поддержка двух или более версий форматов файлов может оказаться непростым делом. Кроме того, функции `json.Encoder.Encode()` и `json.Decoder.Decode()` (а также функции `json.Marshal()` и `json.Unmarshal()`) не являются симметричными друг другу. Это означает, например, что данные можно преобразовать в формат JSON так, что невозможно будет выполнить обратное преобразование и точно восстановить оригинальные данные. Поэтому всегда необходимо проверять их корректную работу на конкретных данных.

Между прочим, существует JSON-подобный формат, который называется BSON (Binary JSON – двоичный JSON), более компактный, чем JSON, и более быстрый. Пакет поддержки формата BSON (gobson) для языка Go доступен на веб-странице godashboard.appspot.com/project. (Установка и использование сторонних пакетов рассматриваются в главе 9.)

8.1.2. Обработка файлов в формате XML

Формат XML (eXtensible Markup Language – расширяемый язык разметки) широко используется как формат обмена данными и как самостоятельный формат файлов. Формат XML более сложный и менее компактный, чем JSON, а реализация его обработки вручную слишком утомительна.

Подобно пакету `encoding/json`, пакет `encoding/xml` позволяет преобразовывать структуры в формат XML и обратно. Однако функции преобразования данных в формат XML и обратно намного более требовательны, чем аналогичные функции в пакете `encoding/json`. Отчасти это обусловлено тем, что пакет `encoding/xml` требует, чтобы поля структур имели соответствующие *теги* форматирования (которые не всегда необходимы для работы с форматом JSON). Кроме того, в версии Go 1 пакет `encoding/xml` не имеет интерфейса `xml.Marshaler`, поэтому для обработки формата XML приходится писать больше программного кода, чем для обработки формата JSON или двоичного формата Go. (Предполагается, что эта проблема будет решена в одной из следующих версий Go 1.x.)

Ниже приводится пример единственной накладной в формате XML, куда были добавлены переводы строк и дополнительные пробелы, чтобы уместить текст по ширине книжной страницы и упростить его восприятие человеком.

```
<INVOICE Id="2640" CustomerId="968" Raised="2012-08-27" Due="2012-09-26"
  Paid="false"><NOTE>See special Terms & Conditions</NOTE>
  <ITEM Id="MI2419" Price="342.80" Quantity="1"><NOTE></NOTE></ITEM>
  <ITEM Id="OU5941" Price="448.99" Quantity="3"><NOTE>
    &quot;Blue&quot; ordered but will accept &quot;Navy&quot;</NOTE>
  </ITEM>
  <ITEM Id="IF9284" Price="475.01" Quantity="1"><NOTE></NOTE></ITEM>
  <ITEM Id="TI4394" Price="417.79" Quantity="2"><NOTE></NOTE></ITEM>
  <ITEM Id="VG4325" Price="80.67" Quantity="5"><NOTE></NOTE></ITEM>
</INVOICE>
```

Возможность появления в тегах символов, имеющих специальное значение в атрибутах XML (например, в полях `Note` накладных и их пунктов), несколько осложняет их обработку с помощью функций преобразования из пакета `xml`, поэтому для представления этих полей используются самостоятельные теги `<NOTE>`.

8.1.2.1. Запись файлов в формате XML

Пакет `encoding/xml` требует, чтобы поля структур имели специализированные теги. По этой причине имеющиеся структуры `Invoice` и `Item` нельзя использовать непосредственно для преобразования в формат XML и обратно. Чтобы решить проблему, были созданы специализированные структуры `XMLInvoices`, `XMLInvoice` и `XMLItem`. А так как программа `invoicedata` требует наличия множества параллельных структур, необходимо также предусмотреть механизмы преобразования между ними. Разумеется, в приложениях, где XML является основным форматом представления данных, достаточно определить только одну структуру (или одно множество структур) и добавить все необходимые теги непосредственно в эти структуры.

Ниже приводится определение структуры `XMLInvoices`, которая будет использоваться для хранения всего набора данных.

```
type XMLInvoices struct {  
    XMLName xml.Name    `xml:"INVOICES"`  
    Version int          `xml:"version,attr"`  
    Invoice []*XMLInvoice `xml:"INVOICE"`  
}
```

Теги полей структур в языке Go не имеют какой-либо определенной семантики – это всего лишь строки, доступные во время выполнения с помощью интерфейса рефлексии (§9.4.9). Однако пакет `encoding/xml` требует определять эти *теги*, передавая в них информацию о том, как будут отображаться поля структур – в формат XML или обратно. Поле `xml.Name` используется для определения имени тега, который будет содержать структуру с этим полем. Поля с тегами ``xml:"",attr`` превратятся в атрибуты тега, где имена полей станут именами атрибутов. При желании можно принудительно использовать другое имя, добавив его перед запятой. Этот прием применен в определении поля `Version`, которое превратится в атрибут с именем `version` вместо имени по умолчанию `Version`. Если тег содержит только имя, это имя будет использоваться в качестве имен вложенных тегов, как тег `<INVOICE>` в данном примере. Важно также

отметить, что поле со списком накладных в структуре XMLInvoices было названо не Invoices, а Invoice, — чтобы обеспечить совпадение (без учета регистра символов) с именем тега.

Ниже приводятся оригинальная структура Invoice и ее XML-эквивалент — структура XMLInvoice.

type Invoice struct {	type XMLInvoice struct {
Id int	XMLName xml.Name `xml:"INVOICE"`
CustomerId int	Id int `xml:",attr"`
Raised time.Time	CustomerId int `xml:",attr"`
Due time.Time	Raised string `xml:",attr"`
Paid bool	Due string `xml:",attr"`
Note string	Paid bool `xml:",attr"`
Items []*Item	Note string `xml:"NOTE"`
	Item []*XMLItem `xml:"ITEM"`
}	}

Для атрибутов здесь использованы имена по умолчанию, например поле CustomerId превратится в атрибут с тем же именем. Здесь также определены два вложенных тега, <NOTE> и <ITEM>, и точно так же, как в случае с полем Invoice в структуре XMLInvoices, чтобы обеспечить совпадение (без учета регистра символов) с именем тега, поле с пунктами накладной получило имя Item вместо Items.

Поля Raised и Due в структуре XMLInvoice объявлены строковыми, потому что программа сама будет обрабатывать их значения (сохраняя только даты) и не будет использовать пакет encoding/xml для сохранения строк с полной датой/временем.

Ниже приводятся оригинальная структура Item и ее XML-эквивалент — структура XMLItem.

type Item struct {	type XMLItem struct {
Id string	XMLName xml.Name `xml:"ITEM"`
Price float64	Id string `xml:",attr"`
Quantity int	Price float64 `xml:",attr"`
Note string	Quantity int `xml:",attr"`
	Note string `xml:"NOTE"`
}	}

Поля структуры XMLItem помечены тегами, превращающими их в атрибуты. Исключение составляют поле Note, которое превратится во вложенный тег <NOTE>, и поле типа XMLName, хранящее имя XML-тега пункта.

Как и в реализации поддержки формата JSON, для поддержки формата XML создана пустая структура с целью определить специализированные методы `MarshalInvoices()` и `UnmarshalInvoices()`.

```
type XMLMarshaler struct{}
```

Этот тип реализует обобщенные интерфейсы `InvoicesMarshaler` и `InvoicesUnmarshaler`, представленные выше (§8.1).

```
func (XMLMarshaler) MarshalInvoices(writer io.Writer,
    invoices []*Invoice) error {
    if _, err := writer.Write([]byte(xml.Header)); err != nil {
        return err
    }
    xmlInvoices := XMLInvoicesForInvoices(invoices)
    encoder := xml.NewEncoder(writer)
    return encoder.Encode(xmlInvoices)
}
```

Данный метод принимает значение типа `io.Writer` (то есть значение любого типа, реализующего интерфейс `io.Writer`, такое как открытый файл или открытый сжатый файл), куда можно выполнить запись данных в формате XML. Метод начинается с записи стандартного заголовка XML `<?xml version="1.0" encoding="UTF-8"?>` (константа `xml.Header` также включает завершающий символ перевода строки). Затем он преобразует все накладные и входящие в них пункты в эквивалентные XML-структуры. На первый взгляд может показаться, что все эти операции будут потреблять такой же объем памяти, какой занимают оригинальные данные, однако, из-за того что строки в языке Go являются неизменяемыми, в действительности будут копироваться лишь ссылки на оригинальные строки, поэтому накладные расходы не так велики, как могло бы показаться. А в приложениях, непосредственно использующих структуры с XML-тегами в полях, такое преобразование вообще не требуется.

После заполнения значения `xmlInvoices` (типа `XMLInvoices`) создается новое значение типа `xml.Encoder`, обертывающее значение типа `io.Writer`, куда должна выполняться запись. Затем все данные преобразуются в формат XML, и вызывающей программе возвращается значение, полученное в результате вызова метода `encoder.Encode()` — признак ошибки или `nil`.

```

func XMLInvoicesForInvoices(invoices []*Invoice) *XMLInvoices {
    xmlInvoices := &XMLInvoices{
        Version: fileVersion,
        Invoice: make([]*XMLInvoice, 0, len(invoices)),
    }
    for _, invoice := range invoices {
        xmlInvoices.Invoice = append(xmlInvoices.Invoice,
            XMLInvoiceForInvoice(invoice))
    }
    return xmlInvoices
}

```

Эта функция принимает значение типа `[]*Invoice` и возвращает указатель `*XMLInvoices` на значение, содержащее все преобразованные данные в виде значений типа `*XMLInvoice` (включающие также значения типа `*XMLItems`), причем практически вся работа выполняется функцией `XmlInvoiceForInvoice()`.

Поле `xml.Name` никогда не должно заполняться программой (если только в программе не используются пространства имен), поэтому здесь при создании значения типа `*XMLInvoices` достаточно лишь заполнить поле `Version`, чтобы обеспечить наличие атрибута `version` в теге `<INVOICES>`, например `<INVOICES version="100">`. Кроме того, для поля `Invoice` здесь создается пустой срез, имеющий достаточную емкость для сохранения всех накладных. Строго говоря, это необязательно, но обеспечивает более высокую эффективность, в сравнении со случаем, если поле оставить с нулевым значением `nil` по умолчанию, потому что при достаточной емкости среза встроенной функции `append()` никогда не придется выделять память и копировать данные по мере увеличения среза.

```

func XMLInvoiceForInvoice(invoice *Invoice) *XMLInvoice {
    xmlInvoice := &XMLInvoice{
        Id:         invoice.Id,
        CustomerId: invoice.CustomerId,
        Raised:      invoice.Raised.Format(dateFormat),
        Due:         invoice.Due.Format(dateFormat),
        Paid:        invoice.Paid,
        Note:        invoice.Note,
        Item:        make([]*XMLItem, 0, len(invoice.Items)),
    }
    for _, item := range invoice.Items {
        xmlItem := &XMLItem{

```



```

        Id:      item.Id,
        Price:   item.Price,
        Quantity: item.Quantity,
        Note:    item.Note,
    }
    xmlInvoice.Item = append(xmlInvoice.Item, xmlItem)
}
return xmlInvoice
}

```

Эта функция принимает значение типа `Invoice` и возвращает эквивалентное ему значение типа `XMLInvoice`. Преобразование выполняется просто: большинство полей структуры `Invoice` просто копируется в соответствующие поля структуры `XMLInvoice`. Поскольку в программе выбран особый подход к преобразованию полей `Raised` и `Due` (сохраняется только дата), они преобразуются в строки. Кроме того, элементы, хранящиеся в поле `Invoice.Items`, добавляются по одному в срез `XMLInvoice.Item` в виде значений типа `XMLItem`. Здесь используется тот же прием оптимизации, что и выше, — создается срез `Item`, обладающий достаточной емкостью, чтобы избежать выделения памяти и копирования данных в функции `append()`. Особенности записи значений типа `time.Time` рассматривались выше, при обсуждении формата JSON (§8.1.1.1).

Напоследок следует обратить внимание, что нигде в программном коде, представленном выше, нам не пришлось *экранировать символы*, имеющие специальное значение в формате XML, — это делается методом `xml.Encoder.Encode()` автоматически.

8.1.2.2. Чтение файлов в формате XML

Чтение XML-файлов реализуется немного сложнее, чем запись, особенно когда некоторые поля (такие как даты) приходится преобразовывать вручную. Но это не сложно, если имеются соответствующие структуры с тегами в полях.

```

func (XMLMarshaler) UnmarshalInvoices(reader io.Reader) ([]*Invoice,
error) {
    xmlInvoices := &XMLInvoices{}
    decoder := xml.NewDecoder(reader)
    if err := decoder.Decode(xmlInvoices); err != nil {
        return nil, err
    }
}

```

```
if xmlInvoices.Version > fileVersion {  
    return nil, fmt.Errorf("version %d is too new to read",  
        xmlInvoices.Version)  
}  
return xmlInvoices.Invoices()  
}
```

Этот метод принимает значение типа `io.Reader` (то есть значение любого типа, реализующего интерфейс `io.Reader`, такое как открытый файл или открытый сжатый файл), откуда будет выполняться чтение данных в формате XML. Метод начинается с создания указателя на пустую структуру `XMLInvoices` и декодера типа `xml.Decoder` для чтения данных из значения типа `io.Reader`. После этого вызовом метода `xml.Decoder.Decode()` выполняется парсинг всего XML-файла, и в случае успеха структура `*XMLInvoices` заполняется данными из XML-файла. Если во время парсинга была обнаружена ошибка (например, синтаксическая ошибка или файл не является файлом с накладными), декодер вернет признак ошибки, который немедленно будет передан вызывающей программе. В случае успеха метод проверяет версию файла, и если она поддерживается программой, все XML-структуры преобразуются во внутреннее представление. Естественно, этого преобразования можно избежать, если изначально использовать структуры с тегами в полях.

```
func (xmlInvoices *XMLInvoices) Invoices() (invoices []*Invoice,  
    err error) {  
    invoices = make([]*Invoice, 0, len(xmlInvoices.Invoice))  
    for _, xmlInvoice := range xmlInvoices.Invoice {  
        invoice, err := xmlInvoice.Invoice()  
        if err != nil {  
            return nil, err  
        }  
        invoices = append(invoices, invoice)  
    }  
    return invoices, nil  
}
```

Метод `XMLInvoices.Invoices()` преобразует значение типа `*XMLInvoices` в значение типа `[]*Invoice`. Он выполняет преобразование, обратное преобразованию, которое производит функция `XmlInvoicesForInvoices()`, и всю работу передает методу `XMLInvoice.Invoice()`.

```
func (xmlInvoice *XMLInvoice) Invoice() (invoice *Invoice, err error) {
    invoice = &Invoice{
        Id:          xmlInvoice.Id,
        CustomerId:  xmlInvoice.CustomerId,
        Paid:        xmlInvoice.Paid,
        Note:       strings.TrimSpace(xmlInvoice.Note),
        Items:      make([]*Item, 0, len(xmlInvoice.Items)),
    }
    if invoice.Raised, err = time.Parse(dateFormat, xmlInvoice.Raised);
        err != nil {
        return nil, err
    }
    if invoice.Due, err = time.Parse(dateFormat, xmlInvoice.Due);
        err != nil {
        return nil, err
    }
    for _, xmlItem := range xmlInvoice.Item {
        item := &Item{
            Id:          xmlItem.Id,
            Price:       xmlItem.Price,
            Quantity:    xmlItem.Quantity,
            Note:        strings.TrimSpace(xmlItem.Note),
        }
        invoice.Items = append(invoice.Items, item)
    }
    return invoice, nil
}
```

Метод `XMLInvoice.Invoice()` используется для преобразования значения типа `*XMLInvoice` в эквивалентное ему значение типа `*Invoice`.

Метод начинается с создания значения типа `Invoice`, большинство полей которого заполняется простым копированием полей из значения типа `XMLInvoice`, где полю `Items` присваивается пустой срез с емкостью, достаточной для хранения всех пунктов накладной.

Затем два поля со значениями даты/времени заполняются вручную, поскольку в программе эти значения обрабатываются вручную. Функция `time.Parse()` принимает строку формата преобразования даты/времени (которая, как отмечалось выше, должна быть основана на точном значении `2006-01-02T15:04:05Z07:00`), и строку для преобразования и возвращает эквивалентное значение типа `time.Time` и `nil` или `nil` и значение ошибки.

Далее в процессе итераций по элементам типа *XMLItems в поле Item структуры XMLInvoice производится создание эквивалентных значений типа *Item и заполнение поля Items. В конце вызывающей программы возвращается значение типа *Invoice.

Так же, как и в реализации записи в XML-файлы, здесь нет необходимости заботиться об обработке *экранированных символов* – метод `xml.Decoder.Decode()` выполнит все необходимые преобразования автоматически.

Пакет `xml` поддерживает намного более сложные теги, чем те, что использовались в данном примере, включая вложенные теги. Например, если для поля структуры определить тег `'xml:"Books>Author"'`, в результате получится XML-тег `<Books><Author>content</Author></Books>`. Кроме того, в дополнение к тегу `'xml:",attr"'` пакет поддерживает тег `'xml:",chardata"'` – для записи значения поля в виде символьных данных, тег `'xml:",innerxml"'` – для записи значения поля без дополнительной интерпретации и тег `'xml:",comment"'` – для записи значения поля в виде XML-комментария. То есть, используя структуры с тегами, можно применять все преимущества удобных функций преобразования и в то же время полностью контролировать все особенности записи и чтения данных в формате XML.

8.1.3. Обработка простых текстовых файлов

В случае применения простых текстовых файлов необходимо определить собственный формат, в идеале как можно более простой для парсинга и расширения в будущем.

Ниже приводится пример записи одной накладной в простом текстовом формате.

```
INVOICE ID=5441 CUSTOMER=960 RAISED=2012-09-06 DUE=2012-10-06 PAID=true
ITEM ID=BE9066 PRICE=400.89 QUANTITY=7: Keep out of <direct> sunlight
ITEM ID=AM7240 PRICE=183.69 QUANTITY=2
ITEM ID=PT9110 PRICE=105.40 QUANTITY=3: Flammable
```



В этом формате информация о каждой накладной записывается в одной строке, начинающейся со слова `INVOICE`, вслед за которой следует одна или более строк с информацией о пунктах этой накладной, начинающихся со слова `ITEM`, и в конце следует символ «перевод формата» (Form Feed). Каждая строка (с информацией о накладной или о пункте) имеет одну и ту же структуру: первым

следует слово, определяющее тип строки, за которым следует последовательность пар *ключ=значение*, затем необязательные двоеточие и текст примечания.

8.1.3.1. Запись файлов в простом текстовом формате

Запись данных в простом текстовом формате реализуется очень просто благодаря мощным и гибким функциям вывода в пакете `fmt`. (Они описывались ранее, в §3.5.)

```

type TxtMarshaler struct{
func (TxtMarshaler) MarshalInvoices(writer io.Writer,
    invoices []*Invoice) error {
    bufferedWriter := bufio.NewWriter(writer)
    defer bufferedWriter.Flush()
    var write writerFunc = func(format string,
        args ...interface{}) error {
        _, err := fmt.Fprintf(bufferedWriter, format, args...)
        return err
    }
    if err := write("%s %d\n", fileType, fileVersion); err != nil {
        return err
    }
    for _, invoice := range invoices {
        if err := write.writeInvoice(invoice); err != nil {
            return err
        }
    }
    return nil
}

```

Этот метод начинается с создания буферизованного значения для записи в файл, переданный методу. Важную роль здесь играет отложенный вызов метода, выталкивающего буфер. Это гарантирует, что все данные фактически будут записаны в файл (если не возникнет ошибка).

Вместо проверки каждой операции записи инструкцией вида `if _, err := fmt.Fprintf(bufferedWriter, ...); err != nil { return err }` здесь создается литерал функции с двумя упрощениями. Во-первых, функция `write()` игнорирует количество записанных байт, возвращаемое функцией `fmt.Fprintf()`. Во-вторых,

функция захватывает значение `bufferedWriter`, благодаря чему отпадает необходимость явно упоминать его в программном коде.

Здесь можно было бы просто написать вспомогательную функцию `writeInvoice(write, invoice)`, принимающую функцию `write()`. Но было решено поступить иначе и добавить методы в тип `writerFunc`. Достигается это простым объявлением методов, принимающих функцию типа `writerFunc` в качестве приемника, по аналогии с методами любых других типов. Этот прием позволяет производить вызовы, такие как `write.writeInvoice(invoice)`, то есть вызывать методы самой функции `write()`. А поскольку такие методы принимают в виде приемника саму функцию `write()`, они легко могут использовать ее.

Обратите внимание, что здесь пришлось явно указать тип (`writerFunc`) функции `write()`. Если этого не сделать, компилятор Go посчитал бы, что она имеет тип `func(string, ...interface{}) error` (какой, собственно, она и имеет) и не позволил бы вызвать метод `writerFunc` (если только не использовать операцию приведения типа в тип `writerFunc`).

Теперь, имея функцию `write()` (с ее методами), можно приступить к записи типа и версии файла (чтобы позднее упростить возможность расширения формата). Затем выполняются итерации по накладным, и для каждой из них вызывается метод `writeInvoice()` функции `write()`.

```
const noteSep = ":"
type writerFunc func(string, ...interface{}) error
func (write writerFunc) writeInvoice(invoice *Invoice) error {
    note := ""
    if invoice.Note != "" {
        note = noteSep + " " + invoice.Note
    }
    if err := write("INVOICE ID=%d CUSTOMER=%d RAISED=%s DUE=%s "+
        "PAID=%t%s\n", invoice.Id, invoice.CustomerId,
        invoice.Raised.Format(dateFormat),
        invoice.Due.Format(dateFormat), invoice.Paid, note); err != nil {
        return err
    }
    if err := write.writeItems(invoice.Items); err != nil {
        return err
    }
    return write("\f\n")
}
```

Этот метод используется для записи каждой накладной. Он принимает накладную для записи и записывает ее с помощью переданной ему функции `write()`.

Строка с информацией о накладной записывается в один прием. Если накладная содержит примечание, перед ним вставляется двоеточие, в противном случае на место примечания вообще ничего не записывается. Для вывода даты/времени (то есть значений типа `time.Time`) используется метод `time.Time.Format()`, как это делалось при записи данных в формате JSON и XML. Для вывода логических значений используется спецификатор формата `%t` (§3.5.1), но точно так же можно было бы использовать спецификатор `%v` или функцию `strconv.FormatBool()` (см. табл. 3.8).

После записи строки с информацией о накладной записываются строки с информацией о пунктах, в конце выводятся символы перевода формата и перевода строки.

```
func (write writerFunc) writeItems(items []*Item) error {
    for _, item := range items {
        note := ""
        if item.Note != "" {
            note = noteSep + " " + item.Note
        }
        if err := write("ITEM ID=%s PRICE=%.2f QUANTITY=%d%s\n", item.Id,
            item.Price, item.Quantity, note); err != nil {
            return err
        }
    }
    return nil
}
```

Метод `writeItems()` принимает пункты накладной и записывает их с помощью функции `write()`, относительно которой он был вызван. Метод выполняет итерации по всем пунктам, записывает их поочередно, и только непустые примечания, как это делается для накладных.

8.1.3.2. Чтение файлов в простом текстовом формате

Чтение данных в простом текстовом формате реализуется практически так же просто, как запись, но парсинг текста с целью восстановить исходные данные может оказаться сложнее, в зависимости от сложности формата.

Существуют четыре основных подхода к реализации парсинга текстовых форматов. Первые три связаны с разбиением строки и последующим использованием функций преобразования, таких как `strconv.Atoi()` и `time.Parse()`, для полей, не являющихся строками. Первый подход основан на выполнении парсинга вручную (например, символ за символом или слово за словом), что может оказаться утомительным занятием, а реализация получается хрупкой и медленной. Второй подход основан на использовании функций `fmt.Fields()` или `fmt.Split()` для разбиения каждой строки на поля. И третий подход основан на использовании регулярных выражений. Для программы `invoicedata` был выбран четвертый подход: он не требует разбивать строки или использовать функции преобразования, потому что практически все, что необходимо, способны выполнить функции сканирования из пакета `fmt`.

```
func (TxtMarshaler) UnmarshalInvoices(reader io.Reader) ([]*Invoice,
    error) {
    bufferedReader := bufio.NewReader(reader)
    if err := checkTxtVersion(bufferedReader); err != nil {
        return nil, err
    }
    var invoices []*Invoice
    eof := false
    for lino := 2; !eof; lino++ {
        line, err := bufferedReader.ReadString('\n')
        if err == io.EOF {
            err = nil // io.EOF в действительности не является ошибкой
            eof = true // это приведет к завершению цикла в следующей итерации
        } else if err != nil {
            return nil, err // в случае действительной ошибки завершить немедленно
        }
        if invoices, err = parseTxtLine(lino, line, invoices); err != nil {
            return nil, err
        }
    }
    return invoices, nil
}
```

Этот метод создает буферизованное значение для переданного ему значения типа `io.Reader` и передает каждую прочитанную строку функции, выполняющей парсинг. Как это обычно для текстовых файлов, возвращаемое значение `io.EOF` обрабатывается

по-особенному, чтобы последняя строка была прочитана всегда, независимо от наличия символа перевода строки в конце. (Конечно, это слишком либеральное допущение для данного конкретного формата.)

Содержимое файла читается построчно, при этом нумерация строк начинается с 1. По первой строке проверяются тип файла и его версия, именно поэтому обработка фактических данных начинается со строки с номером 2 (`lino`).

Поскольку чтение выполняется построчно и каждая накладная представлена двумя или более строками (строка `INVOICE` и одна или более строк `ITEM`), необходимо запоминать текущую накладную, чтобы добавлять к ней пункты по мере чтения последующих строк. В этом нет ничего сложного, потому что накладные добавляются в конец среза `invoices`, то есть текущая накладная всегда будет доступна в позиции `invoices[len(invoices)-1]`.

Когда функция `parseTxtLine()` выполняет парсинг строки `INVOICE`, она создает новое значение типа `Invoice` и добавляет указатель на него в конец среза `invoices`.

Существуют два способа реализации добавления элемента в конец среза внутри функции. Первый: передать указатель на срез и выполнять операции с указателем. Второй: передать срез по значению и возвращать срез (возможно, измененный) обратно, вызывающей программе, где присваивать его оригинальной переменной. Функция `parseTxtLine()` использует второй способ. (Пример использования первого способа был представлен выше, в §5.7.)

```
func parseTxtLine(lino int, line string, invoices []*Invoice) ([]*Invoice,
    error) {
    var err error
    if strings.HasPrefix(line, "INVOICE") {
        var invoice *Invoice
        invoice, err = parseTxtInvoice(lino, line)
        invoices = append(invoices, invoice)
    } else if strings.HasPrefix(line, "ITEM") {
        if len(invoices) == 0 {
            err = fmt.Errorf("item outside of an invoice line %d", lino)
        } else {
            var item *Item
            item, err = parseTxtItem(lino, line)
            items := &invoices[len(invoices)-1].Items ❶
        }
    }
}
```

```

        *items = append(*items, item)
    }
}
return invoices, err
}

```

Эта функция принимает номер строки (`lineno` для вывода сообщения об ошибке), строку для анализа и срез с накладными, который требуется заполнить.

Если строка начинается с текста «INVOICE», вызывается функция `parseTxtInvoice()`, которая выполнит парсинг строки, создаст значение типа `Invoice` и вернет указатель на него. Затем функция добавляет полученное значение типа `*Invoice` в конец среза `invoices` и возвращает его и `nil` или признак ошибки. Обратите внимание, что в этот момент накладная заполнена не полностью – в ней заполнены только поля `Id`, `CustomerId`, `Raised` и `Due`, `Paid` и `Note`, но пока не было заполнено ни одного пункта.

Если строка начинается с текста «ITEM», сначала выполняется проверка наличия текущей накладной (то есть проверяется длина среза `invoices`, которая должна быть больше нуля). Если текущая накладная существует, вызывается функция `parseTxtItem()`, которая выполняет парсинг строки, создает значения типа `Item` и возвращает указатель на него. Полученный пункт необходимо добавить в текущую накладную. Для этого функция получает указатель на поле `items` (❶) текущей накладной с последующим присваиванием указателю значения (типа `[]*Item`), полученного в результате добавления в срез нового элемента типа `*Item`. Добавить новое значение `*Item` можно было бы непосредственно, используя инструкцию `invoices[len(invoices)-1].Items = append(invoices[len(invoices)-1].Items, item)`.

Любые другие строки (пустые строки и строки с символами перевода формата) просто игнорируются. Теоретически эту функцию можно сделать немного быстрее, если проверку случая «ITEM» выполнять первой, потому что пунктов накладных обычно больше, чем самих накладных или пустых строк.

```

func parseTxtInvoice(lino int, line string) (invoice *Invoice,
    err error) {
    invoice = &Invoice{}
    var raised, due string
    if _, err = fmt.Sscanf(line, "INVOICE ID=%d CUSTOMER=%d "+

```

```

    "RAISED=%s DUE=%s PAID=%t", &invoice.Id, &invoice.CustomerId,
    &raised, &due, &invoice.Paid); err != nil {
    return nil, fmt.Errorf("invalid invoice %v line %d", err, lino)
}
if invoice.Raised, err = time.Parse(dateFormat, raised); err != nil {
    return nil, fmt.Errorf("invalid raised %v line %d", err, lino)
}
if invoice.Due, err = time.Parse(dateFormat, due); err != nil {
    return nil, fmt.Errorf("invalid due %v line %d", err, lino)
}
if i := strings.Index(line, noteSep); i > -1 {
    invoice.Note = strings.TrimSpace(line[i+len(noteSep):])
}
return invoice, nil
}

```

Функция начинается с создания нового значения типа `Invoice` и присваивания указателя на него переменной (типа `*Invoice`). Функции сканирования способны обрабатывать строки, числа и логические значения, но они не поддерживают значения типа `time`. `Time`, поэтому даты обрабатываются сканированием их как строк с последующим преобразованием отдельно. Функции сканирования перечислены в табл. 8.2.

Если количество элементов, прочитанных функцией `fmt.Sscanf()`, окажется меньше количества значений, которые требуется заполнить, или возникнет какая-то другая ошибка (например, ошибка чтения), она вернет непустое значение ошибки.

Парсинг дат выполняется с помощью функции `time.Parse()`, обсуждавшейся выше (§8.1.2.2). Если строка с информацией о накладной содержит двоеточие, это означает, что в конце присутствует примечание, которое извлекается с попутным отсечением пробельных символов в начале и в конце. Вместо выражения `line[i+len(noteSep):]` можно было бы использовать выражение `line[i+1:]`, поскольку известно, что символу двоеточия в строке `noteSep` соответствует единственный байт в кодировке UTF-8, но здесь решено было соблюсти меры предосторожности и использовать подход, пригодный для любых символов, не зависящий от количества байт, соответствующих символу.

```

func parseTxtItem(lino int, line string) (item *Item, err error) {
    item = &Item{}
    if _, err = fmt.Sscanf(line, "ITEM ID=%s PRICE=%f QUANTITY=%d",
        &item.Id, &item.Price, &item.Quantity); err != nil {

```

```

    return nil, fmt.Errorf("invalid item %v line %d", err, lino)
}
if i := strings.Index(line, noteSep); i > -1 {
    item.Note = strings.TrimSpace(line[i+1:len(noteSep):])
}
return item, nil
}

```

Таблица 8.2. Функции сканирования из пакета *fmt*

Параметр *r* имеет тип *io.Reader* и используется как значение для чтения. Параметр *s* имеет тип *string* и используется как значение для чтения. Параметр *fs* – это строка формата, используемая функциями вывода в пакете *fmt* (см. табл. 3.4). Параметр *args* – один или более указателей (то есть адресов) на значения, которые требуется заполнить. Все функции сканирования возвращают количество успешно извлеченных элементов и либо *nil*, либо значение ошибки.

Функция	Описание/результат
<code>fmt.Fscan(r, args)</code>	Читает из <i>r</i> последовательность значений, разделенных пробелами или символами перевода строки, для заполнения элементов <i>args</i>
<code>fmt.Fscanf(r, fs, args)</code>	Читает из <i>r</i> последовательность значений, разделенных пробелами, в соответствии с форматом <i>fs</i> для заполнения элементов <i>args</i>
<code>fmt.Fscanln(r, args)</code>	Читает из <i>r</i> последовательность значений, разделенных пробелами, для заполнения элементов <i>args</i> и ожидает в конце встретить <i>io.EOF</i> или символ перевода строки
<code>fmt.Scan(args)</code>	Читает из <i>os.Stdin</i> последовательность значений, разделенных пробелами, для заполнения элементов <i>args</i>
<code>fmt.Scanf(fs, args)</code>	Читает из <i>os.Stdin</i> последовательность значений, разделенных пробелами, в соответствии с форматом <i>fs</i> для заполнения элементов <i>args</i>
<code>fmt.Scanln(args)</code>	Читает из <i>os.Stdin</i> последовательность значений, разделенных пробелами, в соответствии с форматом <i>fs</i> для заполнения элементов <i>args</i> .
<code>fmt.Sscan(s, args)</code>	Читает из <i>s</i> последовательность значений, разделенных пробелами или символами перевода строки, для заполнения элементов <i>args</i>
<code>fmt.Sscanf(s, fs, args)</code>	Читает из <i>s</i> последовательность значений, разделенных пробелами, в соответствии с форматом <i>fs</i> для заполнения элементов <i>args</i>
<code>fmt.Sscanln(s, args)</code>	Читает из <i>s</i> последовательность значений, разделенных пробелами, для заполнения элементов <i>args</i> и ожидает в конце встретить <i>io.EOF</i> или символ перевода строки

Эта функция действует так же, как функция `parseTxtInvoice()`, представленная выше, за исключением того, что в строке с информацией о пункте накладной все элементы, кроме примечания, могут быть извлечены непосредственно.

```
func checkTxtVersion(bufferedReader *bufio.Reader) error {
    var version int
    if _, err := fmt.Fscanf(bufferedReader, "INVOICES %d\n", &version);
        err != nil {
        return errors.New("cannot read non-invoices text file")
    } else if version > fileVersion {
        return fmt.Errorf("version %d is too new to read", version)
    }
    return nil
}
```

Эта функция используется для чтения самой первой строки из текстового файла с накладными. Для чтения из значения типа `bufio.Reader` она использует функцию `fmt.Fscanf()`. Если файл не является файлом с накладными или если файл имеет версию, не поддерживаемую программой, она сообщает об ошибке, в противном случае возвращает `nil`.

Запись в текстовые файлы легко реализуется с помощью функций вывода из пакета `fmt`. Парсинг текстовых файлов может оказаться не простым делом, но в распоряжении программиста имеется пакет `regex`, функции `strings.Fields()` и `strings.Split()` и функции сканирования из пакета `fmt`.

8.1.4. Обработка файлов в двоичном формате Go

Двоичный формат Go («gob») представляет собой последовательность двоичных значений с описанием. Во внутреннем представлении двоичный формат Go состоит из нуля или более фрагментов, каждый из которых содержит счетчик байтов, последовательность из нуля или более пар *ИдентификаторТипа/СпецификацияТипа* и пар *ИдентификаторТипа/Значение*. Пары *ИдентификаторТипа/ОписаниеТипа* могут быть опущены для предопределенных типов (таких как `bool`, `int`, `string` и др.), в противном случае каждая из этих пар используется для описания пользовательского типа (например, структур). Чтобы отличать пары с описанием типа от пар со значениями, числа, определяющие идентификаторы типов в парах с

описанием, инвертируются. Однако, как будет показано далее, чтобы использовать формат `gob`, не требуется знать всех технических подробностей его устройства, потому что обо всех низкоуровневых деталях позаботится пакет `encoding/gob`¹.

Пакет `encoding/gob` содержит функции для преобразования данных в формат `gob` и обратно, которые практически не отличаются от аналогичных функций в пакете `encoding/json` и так же просты в использовании. В целом формат `gob` является наиболее удобным для файлов с данными или для передачи данных через сетевые соединения, когда не требуется обеспечить возможность восприятия этих данных человеком.

8.1.4.1. Запись файлов в двоичном формате *Go*

Следующий метод выполняет запись всего множества элементов среза типа `[]*Invoice` в открытый файл (или в значение любого другого типа, реализующего интерфейс `io.Writer`) в формате `gob`.

```
type GobMarshaler struct{}  
  
func (GobMarshaler) MarshalInvoices(writer io.Writer,  
    invoices []*Invoice) error {  
    encoder := gob.NewEncoder(writer)  
    if err := encoder.Encode(magicNumber); err != nil {  
        return err  
    }  
    if err := encoder.Encode(fileVersion); err != nil {  
        return err  
    }  
    return encoder.Encode(invoices)  
}
```

Сначала метод создает значение для преобразования данных в формат `gob`, которое оборачивает значение, реализующее интерфейс `io.Writer`, и получает значение, куда будет выполняться запись.

Запись данных производится вызовом метода `gob.Encoder.Encode()`. Он прекрасно справляется с накладными, каждая из которых содержит собственный список пунктов. Метод возвращает значение `nil`

¹ Более подробное описание формата приводится в документации, на странице golang.org/pkg/encoding/gob/. Интересно будет также ознакомиться со статьей Роба Пайка (Rob Pike) о формате `gob`, написанной им в блоге, посвященном программированию на языке Go: blog.golang.org/2011/03/gobs-of-data.html.

или признак ошибки. В случае ошибки она немедленно возвращается вызывающей программе.

Запись сигнатуры и версии файла в принципе необязательна, но, как будет показано в одном из упражнений, их наличие упростит изменение формата файла в будущем.

Обратите внимание, что метод никак не учитывает типов записываемых им данных, поэтому совсем несложно будет создать аналогичные функции для записи данных в формате `gob`. Кроме того, метод `GobMarshaler.MarshalInvoices()` не потребуется изменять при изменении формата файла.

Поскольку все поля структуры `Invoice` являются логическими, числовыми, строковыми значениями, значениями типа `time.Time` или составными значениями (поле `Items`), которые сами содержат только логические, числовые, строковые значения, значения типа `time.Time` или составные значения, представленный программный код успешно справляется с их преобразованием.

Если бы структура содержала данные, не поддерживающие преобразования в формат `gob`, для нее пришлось бы реализовать интерфейсы `gob.GobEncoder` и `gob.GobDecoder`. В процессе преобразования в формат `gob` автоматически проверяется поддержка интерфейса `gob.GobEncoder`, и при ее наличии вместо встроенной процедуры преобразования используется метод `GobEncode()` преобразуемого значения. То же относится и к процедуре обратного преобразования, где проверяется наличие метода `GobDecode()`, определяемого интерфейсом `gob.GobDecoder`. (В исходный файл `gob.go` примера `invoicedata` включены – ненужные и закомментированные – методы преобразования значения типа `Invoice` в формат `gob` и обратно, исключительно чтобы показать, насколько просто они реализуются.) Реализация этих интерфейсов в структуре может существенно замедлить запись и чтение данных в формате `gob` и привести к увеличению размеров файлов.

8.1.4.2. Чтение файлов в двоичном формате Go

Чтение данных в формате `gob` выполняется так же просто, как и запись, особенно если чтение выполняется в переменные того же типа, что и переменные, откуда выполнялась запись. Метод `GobMarshaler.UnmarshalInvoices()` принимает значение с поддержкой интерфейса `io.Reader` (например, файл, открытый для чтения), откуда требуется прочитанные данные в формате `gob`.

```
func (GobMarshaler) UnmarshalInvoices(reader io.Reader) ([]*Invoice,
    error) {
    decoder := gob.NewDecoder(reader)
    var magic int
    if err := decoder.Decode(&magic); err != nil {
        return nil, err
    }
    if magic != magicNumber {
        return nil, errors.New("cannot read non-invoices gob file")
    }
    var version int
    if err := decoder.Decode(&version); err != nil {
        return nil, err
    }
    if version > fileVersion {
        return nil, fmt.Errorf("version %d is too new to read", version)
    }
    var invoices []*Invoice
    err := decoder.Decode(&invoices)
    return invoices, err
}
```

Программе требуется прочитать три элемента: сигнатуру файла, его версию и все накладные. Метод `gob.Decoder.Decode()` принимает указатель на значение, которое требуется заполнить, и возвращает признак ошибки или `nil`. Первые две переменные (`magic` и `version`) используются, чтобы убедиться, что файл действительно содержит список накладных, а его версия поддерживается программой. Затем выполняется чтение накладных, в процессе которого метод `gob.Decoder.Decode()` увеличивает размер среза `invoices` и заполняет его указателями на значения типа `Invoice` (и их пункты типа `Item`), создаваемые по мере необходимости. В конце метод возвращает срез `invoices` и `nil` или признак ошибки, если будет обнаружена какая-либо проблема.

Если структура накладной изменится за счет добавления новых экспортируемых полей, метод с успехом будет работать в текущем своем виде, если новые поля будут содержать логические, числовые, строковые значения, значения типа `time.Time` или структуры, содержащие эти типы. Разумеется, если в структуре появятся поля других типов, необходимо будет привести в соответствие методы, реализующие интерфейсы `gob.GobEncoder` и `gob.GobDecoder`.

Формат `gob` обладает очень большой гибкостью при работе с составными типами, легко справляясь с некоторыми различиями.

Например, если структуру, имеющую некоторое значение, записать в формате `gob`, это значение можно будет прочитать обратно в ту же самую структуру или в похожие структуры, возможно, содержащие указатель на значение, или где тип значения отличается, но совместим с первоначальным (например, `int` и `uint`). И, как показывает пример `invoicedata`, формат `gob` с легкостью справляется с вложенными данными (однако пока он не способен обрабатывать рекурсивные значения). В описании формата `gob` обсуждаются различия, с которыми он способен обрабатывать, и разъясняется его внутреннее устройство. Однако к данному примеру это не относится, потому что для записи и чтения используются значения одних и тех же типов.

8.1.5. Обработка файлов в пользовательском двоичном формате

Несмотря на то что пакет `encoding/gob` очень прост в использовании и для работы с ним требуется написать совсем немного программного кода, иногда может все же потребоваться реализовать собственный *двоичный формат* хранения данных. Пользовательский формат может создаваться для достижения максимальной компактности и очень высокой скорости чтения и записи. На практике мне не раз приходилось убеждаться, что скорость чтения и записи двоичного формата `Go` существенно выше скорости чтения и записи пользовательских двоичных форматов, а размеры файлов получаются немногим больше. Однако некоторые из этих преимуществ утрачиваются, когда возникает необходимость обрабатывать данные, не поддерживающие возможности преобразования в формат `gob`, и реализовать интерфейсы `gob.GobEncoder` и `gob.GobDecoder`. В некоторых ситуациях может также потребоваться обмениваться данными с другими приложениями, использующими собственные двоичные форматы, поэтому знание, как обрабатывать двоичные файлы, может оказаться весьма полезным.

На рис. 8.1 изображено схематическое представление одной накладной в `inv`. Целочисленные значения представлены целочисленными типами определенных размеров со знаком или без знака. Логические значения представлены типом `int8`, где значение 1 соответствует значению `true`, а 0 – значению `false`. Строки представлены счетчиком байтов (типа `int32`), за которым следует срез типа `[]byte` с байтами в кодировке UTF-8. Для дат был выбран немного необычный подход – они представлены значениями типа `int32` на

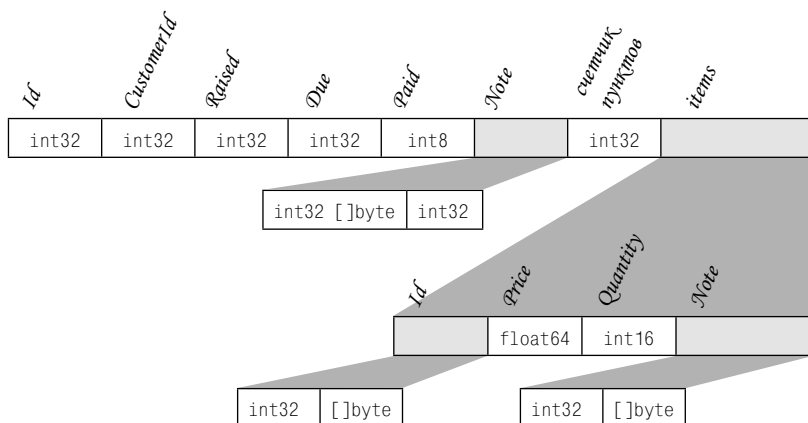


Рис. 8.1. Пользовательский двоичный формат *.inv*

основе формата ISO-8601 представления дат (без дефисов) в виде числа. Например, дата 2006-01-02 будет представлена как целое число 20060102. Пункты накладной представлены счетчиком пунктов, за которым следуют сами пункты. (Напомним, что, в отличие от идентификаторов накладных, идентификаторы пунктов являются строками, а не целыми числами, см. §8.1.)

8.1.5.1. Запись файлов в пользовательском двоичном формате

Запись данных в двоичном формате легко реализуется благодаря функции `binary.Write()`, имеющейся в пакете `encoding/binary`.

```

type InvMarshaler struct{}
var byteOrder = binary.LittleEndian
func (InvMarshaler) MarshalInvoices(writer io.Writer,
    invoices []*Invoice) error {
    var write invWriterFunc = func(x interface{}) error {
        return binary.Write(writer, byteOrder, x)
    }
    if err := write(uint32(magicNumber)); err != nil {
        return err
    }
    if err := write(uint16(fileVersion)); err != nil {
        return err
    }
    if err := write(int32(len(invoices))); err != nil {

```

```

    return err
}
for _, invoice := range invoices {
    if err := write.writeInvoice(invoice); err != nil {
        return err
    }
}
return nil
}

```

Этот метод записывает все накладные в указанное значение типа `io.Writer`. Он начинается с создания вспомогательной функции `write()`, которая захватывает и значение типа `io.Writer`, и переменную `byteOrder`, определяющую порядок следования байт. Так же, как это было реализовано для простого текстового формата, функция `write()` определена как функция определенного типа (`invWriterFunc`), и для нее было создано несколько методов (например, `invWriterFunc.writeInvoices()`), которые пригодятся ниже.

Обратите внимание: при чтении и записи двоичных данных *очень важно* использовать один и тот же *порядок следования байтов*. (Здесь невозможно определить константу вместо переменной `byteOrder`, потому что `binary.LittleEndian` и `binary.BigEndian` не являются простыми значениями, такими как строки или числа.)

Запись данных в пользовательском двоичном формате очень напоминает запись данных в других форматах. Однако имеется одно важное отличие, которое состоит в том, что после записи сигнатуры и версии файла необходимо записать количество накладных. (Можно было бы опустить этот счетчик и просто записывать накладные одну за другой, а затем при чтении просто читать накладные до достижения признака `io.EOF`.)

```

type invWriterFunc func(interface{}) error
func (write invWriterFunc) writeInvoice(invoice *Invoice) error {
    for _, i := range []int{invoice.Id, invoice.CustomerId} {
        if err := write(int32(i)); err != nil {
            return err
        }
    }
    for _, date := range []time.Time{invoice.Raised, invoice.Due} {
        if err := write.writeDate(date); err != nil {
            return err
        }
    }
}

```

```

    }
    if err := write.writeBool(invoice.Paid); err != nil {
        return err
    }
    if err := write.writeString(invoice.Note); err != nil {
        return err
    }
    if err := write.writeInt32(len(invoice.Items)); err != nil {
        return err
    }
    for _, item := range invoice.Items {
        if err := write.writeItem(item); err != nil {
            return err
        }
    }
    return nil
}

```

Метод `writeInvoice()` вызывается для записи каждой накладной. Он принимает указатель на структуру с накладной и записывает ее с помощью переданной ему функции `write()`.

Метод начинается с записи полей `Id` и `CustomerId` в виде значений типа `int32`. Эти значения можно было бы записать как значения типа `int`, но это непереносимо, потому что размер типа `int` может изменяться в зависимости от аппаратной архитектуры и используемой версии Go. Поэтому очень важно всегда использовать целочисленные типы со знаком или без знака, определенного размера, такие как `uint32`, `int32`, и т. д. Далее выполняется запись дат из полей `Raised` и `Due` с использованием собственного метода `writeDate()`, затем логического поля `Paid` и строкового поля `Note`, опять же с использованием собственных методов. Наконец, записывается счетчик пунктов накладной, за которым следуют сами пункты, каждый из которых записывается собственным методом `writeItem()`.

```

const invDateFormat = "20060102" // Всегда должна использоваться эта дата.
func (write invWriterFunc) writeDate(date time.Time) error {
    i, err := strconv.Atoi(date.Format(invDateFormat))
    if err != nil {
        return err
    }
    return write.writeInt32(i)
}

```

Функция `time.Time.Format()` уже обсуждалась, и там же говорилось, почему необходимо использовать конкретную дату `2006-01-02` в строках формата (§8.1.1.1). Здесь используется формат, напоминающий формат ISO-8601, но без дефисов, поэтому в результате получается строка, содержащая точно восемь цифр с ведущими нулями в номерах и числах месяца, состоящих из одной цифры. Затем эта строка преобразуется в целое число, например для даты `2012-08-05` будет получено число `20120805`, которое записывается как значение типа `int32`.

В случае если бы потребовалось сохранять не только дату, но еще и время, или просто для ускорения вычислений, вызовы этого метода можно было бы заменить вызовом `write(int64(date.Unix()))` и сохранять время в виде количества секунд, прошедших с начала эпохи Unix. Соответствующая операция чтения могла бы быть реализована так: `var d int64; if err := binary.Read(reader, byteOrder, &d); err != nil { return err }; date := time.Unix(d, 0)..`

```
func (write invWriterFunc) writeBool(b bool) error {  
    var v int8  
    if b {  
        v = 1  
    }  
    return write(v)  
}
```

На момент написания этих строк пакет `encoding/binary` не поддерживал чтения и записи логических значений, поэтому для их обработки был создан этот простой метод. Кстати, при записи нет необходимости использовать операцию преобразования типа (такую как `int8(v)`), потому что переменная `v` уже имеет тип фиксированного размера.

```
func (write invWriterFunc) writeString(s string) error {  
    if err := write(int32(len(s))); err != nil {  
        return err  
    }  
    return write([]byte(s))  
}
```

Строки должны записываться в виде последовательностей составляющих их байт в кодировке UTF-8. Здесь сначала записывается

счетчик байтов, а затем сами байты. (Если бы все строки имели фиксированный размер, записывать счетчик не потребовалось бы, потому что при чтении можно было бы просто создать пустой срез `[]byte` того же размера.)

```
func (write invWriterFunc) writeItem(item *Item) error {
    if err := write.writeString(item.Id); err != nil {
        return err
    }
    if err := write(item.Price); err != nil {
        return err
    }
    if err := write(int16(item.Quantity)); err != nil {
        return err
    }
    return write.writeString(item.Note)
}
```

Этот метод вызывается для записи каждого пункта в каждой накладной. Запись строковых полей `Id` и `Note` выполняется вызовом метода `invWriterFunc.writeString()`. Поле `Quantity` записывается как значение типа с фиксированным размером. Однако для записи поля `Price` не требуется преобразовывать тип его значения, потому что оно и так имеет тип фиксированного размера (`float64`).

В реализации записи двоичных данных нет ничего сложного, достаточно лишь позаботиться о сохранении счетчиков перед данными, имеющими переменную длину, чтобы потом можно было определить, сколько элементов требуется прочитать. Формат `gob`, безусловно, более удобный, но применение пользовательского двоичного формата позволяет уменьшить размеры файлов.

8.1.5.2. Чтение файлов в пользовательском двоичном формате

Чтение данных в пользовательском двоичном формате реализуется так же просто, как и их запись. Здесь не требуется выполнять парсинг данных – достаточно просто читать их, используя тот же порядок следования байтов, что и при записи, в значения того же типа.

```
func (InvMarshaler) UnmarshalInvoices(reader io.Reader) ([]*Invoice,
error) {
    if err := checkInvVersion(reader); err != nil {
```

```

        return nil, err
    }
    count, err := readIntFromInt32(reader)
    if err != nil {
        return nil, err
    }
    invoices := make([]*Invoice, 0, count)
    for i := 0; i < count; i++ {
        invoice, err := readInvInvoice(reader)
        if err != nil {
            return nil, err
        }
        invoices = append(invoices, invoice)
    }
    return invoices, nil
}

```

Этот метод начинается с проверки сигнатуры и версии файла. Затем он читает количество накладных, имеющихся в файле, используя функцию `readIntFromInt32()`. Далее создается срез с нулевой длиной (то есть без накладных), но с требуемой емкостью. Затем функция читает накладные одну за другой и добавляет их в срез `invoices`.

Вызов функции `make()` можно было бы заменить вызовом `make([]*invoice, count)`, а вызов функции `append()` – инструкцией `invoices[i] = invoice`. Однако предпочтительнее создавать срезы необходимой емкости, поскольку добавление в них новых элементов выполняется быстрее, чем добавление с сопутствующим увеличением размера среза. В конце концов, если выполнять добавление в срез, длина которого равна его емкости, всякий раз будет создаваться новый срез, и в него будут копироваться данные из старого среза, тогда как при достаточной емкости копирование не производится.

```

func checkInvVersion(reader io.Reader) error {
    var magic uint32
    if err := binary.Read(reader, byteOrder, &magic); err != nil {
        return err
    }
    if magic != magicNumber {
        return errors.New("cannot read non-invoices inv file")
    }
    var version uint16
    if err := binary.Read(reader, byteOrder, &version); err != nil {

```

```

    return err
}
if version > fileVersion {
    return fmt.Errorf("version %d is too new to read", version)
}
return nil
}

```

Эта функция пытается прочитать сигнатуру и версию файла. Она возвращает `nil`, если можно продолжить чтение файла, и признак ошибки в противном случае.

Функция `binary.Read()` является парной для функции `binary.Write()` — она принимает значение типа `io.Reader`, откуда будет выполняться чтение, порядок следования байтов и указатель на значение требуемого типа, куда должно быть выполнено чтение.

```

func readIntFromInt32(reader io.Reader) (int, error) {
    var i32 int32
    err := binary.Read(reader, byteOrder, &i32)
    return int(i32), err
}

```

Эта вспомогательная функция используется для чтения значения типа `int32` из двоичного файла и возвращает его в виде значения типа `int`.

```

func readInvInvoice(reader io.Reader) (invoice *Invoice, err error) {
    invoice = &Invoice{}
    for _, pId := range []int{&invoice.Id, &invoice.CustomerId} {
        if *pId, err = readIntFromInt32(reader); err != nil {
            return nil, err
        }
    }
    for _, pDate := range []time.Time{&invoice.Raised, &invoice.Due} {
        if *pDate, err = readInvDate(reader); err != nil {
            return nil, err
        }
    }
    if invoice.Paid, err = readBoolFromInt8(reader); err != nil {
        return nil, err
    }
    if invoice.Note, err = readInvString(reader); err != nil {
        return nil, err
    }
}

```



```
var count int
if count, err = readIntFromInt32(reader); err != nil {
    return nil, err
}
invoice.Items, err = readInvItems(reader, count)
return invoice, err
}
```

Эта функция вызывается для чтения каждой накладной. Сначала она создает новую пустую накладную и сохраняет указатель на нее в переменной `invoice`.

Чтение полей `Id` и `CustomerId` выполняется с помощью функции `readIntFromInt32()`. Необычность ситуации – в том, что здесь выполняются итерации по указателям на поля `Id` и `CustomerId` и присваивание значения типа `int`, возвращаемого функцией, производится по указателю (`pId`).

Каждое из этих двух полей можно было бы обрабатывать по отдельности, например: `if invoice.Id, err = readIntFromInt32(reader); err != nil { return err }` и т. д.

Чтение полей `Raised` и `Due` выполняется по тому же шаблону, что и чтение двух предыдущих полей, только на этот раз используется функция `readInvDate()`.

Как и в случае с двумя первыми полями, поля `Raised` и `Due` можно было бы обрабатывать по отдельности, например: `if invoice.Due, err = readInvDate(reader); err != nil { return err }` и т. д.

Чтение значений логического поля `Paid` и строкового поля `Note` выполняется с помощью вспомогательных функций, которые будут представлены чуть ниже. После накладной программа читает счетчик пунктов, а затем с помощью функции `readInvItems()` читает сами пункты, передавая функции значение типа `io.Reader`, откуда должно выполняться чтение, и количество пунктов.

```
func readInvDate(reader io.Reader) (time.Time, error) {
    var n int32
    if err := binary.Read(reader, byteOrder, &n); err != nil {
        return time.Time{}, err
    }
    return time.Parse(invDateFormat, fmt.Sprint(n))
}
```

Эта функция читает представление даты в виде значения типа `int32` (например, `20130501`), затем выполняет парсинг строкового

представления этого числа и возвращает соответствующее значение типа `time.Time` (например, 2013-05-01).

```
func readBoolFromInt8(reader io.Reader) (bool, error) {
    var i8 int8
    err := binary.Read(reader, byteOrder, &i8)
    return i8 == 1, err
}
```

Эта простая вспомогательная функция возвращает `true`, если прочитанное значение типа `int8` равно 1, и `false` – в противном случае.

```
func readInvString(reader io.Reader) (string, error) {
    var length int32
    if err := binary.Read(reader, byteOrder, &length); err != nil {
        return "", nil
    }
    raw := make([]byte, length)
    if err := binary.Read(reader, byteOrder, &raw); err != nil {
        return "", err
    }
    return string(raw), nil
}
```

Эта функция читает байты в срез типа `[]byte`, но принцип, используемый здесь, с успехом можно применить для чтения срезов любых типов, если при записи перед ними сохранялся счетчик содержащихся в них элементов.

Сначала в переменную `length` читается счетчик элементов. Затем создается срез этой длины. Когда функция `binary.Read()` получает указатель на срез, она читает столько элементов, тип которых соответствует типу среза, сколько поместится в срез (или терпит неудачу и возвращает непустой признак ошибки). Обратите внимание, что здесь имеет значение *длина* среза, а не его емкость (которая может быть равна длине или превышать ее).

В данном случае срез типа `[]byte` хранит байты, представляющие символы в кодировке UTF-8, и при возврате он преобразуется в строку.

```
func readInvItems(reader io.Reader, count int) ([]*Item, error) {
    items := make([]*Item, 0, count)
    for i := 0; i < count; i++ {
        item, err := readInvItem(reader)
        if err != nil {
```

```
        return nil, err
    }
    items = append(items, item)
}
return items, nil
}
```

Эта функция читает все пункты накладной. В аргументе `count` она получает количество пунктов, которые требуется прочитать.

```
func readInvItem(reader io.Reader) (item *Item, err error) {
    item = &Item{}
    if item.Id, err = readInvString(reader); err != nil {
        return nil, err
    }
    if err = binary.Read(reader, byteOrder, &item.Price); err != nil {
        return nil, err
    }
    if item.Quantity, err = readIntFromInt16(reader); err != nil {
        return nil, err
    }
    item.Note, err = readInvString(reader)
    return item, nil
}
```

Эта функция читает один пункт накладной. По своей структуре она очень похожа на функцию `readInvInvoice()` тем, что создает пустое значение типа `Item`, сохраняет указатель на него в переменной `item` и затем заполняет поля структуры. Значение для поля `Price` можно читать непосредственно, потому что оно было записано как значение типа `float64`, то есть типа с фиксированным размером. (Здесь не представлена функция `readIntFromInt16()`, потому что она практически идентична функции `readIntFromInt32()`, представленной выше.)

На этом завершается обзор особенностей чтения и записи данных в пользовательском двоичном формате. Работать с двоичными данными совсем несложно, если использовать типы фиксированного размера и предварять счетчиками данные переменной длины (такие как срезы).

Поддержка двоичных файлов в языке Go включает возможность произвольного доступа. В таких случаях открывать файл следует с помощью функции `os.OpenFile()` (а не `os.Open()`) и передавать ей

соответствующие флаги разрешений и режимов (например, `os.O_RDONLY` для доступа на чтение и запись)¹. Затем можно использовать метод `os.File.Seek()` для позиционирования в файле перед чтением и записью или методы `os.File.ReadAt()` и `os.File.WriteAt()` для чтения и записи с определенной позиции в файле. Имеется также ряд других полезных методов, включая `os.File.Stat()`, который возвращает структуру типа `os.FileInfo` с информацией о размере файла, правах доступа к нему, а также с датами и временами.

8.2. Архивные файлы

В стандартной библиотеке языка Go реализована поддержка нескольких форматов сжатия, в том числе и формата `gzip`, благодаря чему в программах на языке Go легко можно организовать прозрачную поддержку файлов с `gzip`-сжатием, если их имена имеют расширение `.gz`, и без сжатия в противном случае. Кроме того, в библиотеке имеются пакеты, позволяющие писать и читать `zip`-файлы и тарболлы (с расширениями `.tar` и `.tar.gz`), а также читать файлы `.bz2` (обычно с расширением `.tar.bz2`).

В этом разделе рассматриваются выдержки из двух программ. Первая программа, `pack` (в файле `pack/pack.go`), принимает в виде аргументов командной строки имя будущего архива и список имен файлов для архивирования. Формат файла определяется в ней на основе расширения имени архива. Вторая программа, `unpack` (в файле `unpack/unpack.go`), также принимает имя архива и пытается извлечь из него все файлы, воссоздавая структуру дерева каталогов при необходимости.

8.2.1. Создание `zip`-архивов

Чтобы использовать пакет `zip` для сжатия, необходимо сначала открыть файл для записи и затем создать значение типа `*zip.Writer` для записи в него. Далее для каждого файла, который требуется поместить в `zip`-архив, нужно прочитать его содержимое и записать в

¹ Флаги разрешений традиционно записываются в виде восьмеричных чисел, начинающихся с ведущего 0. Значение 0666 обеспечивает возможность чтения и записи файла для любого пользователя, однако значение *umask*, равное 0022 (широко используемое значение), превращает его в 0644, что делает файл доступным для чтения и записи пользователю, создавшему его, и только для чтения всем остальным.

значение типа `io.Writer`, полученное из значения `*zip.Writer`. Для создания zip-архивов подобным способом программа `pack` использует две функции, `createZip()` и `writeFileToZip()`.

```
func createZip(filename string, files []string) error {
    file, err := os.Create(filename)
    if err != nil {
        return err
    }
    defer file.Close()
    zipper := zip.NewWriter(file)
    defer zipper.Close()
    for _, name := range files {
        if err := writeFileToZip(zipper, name); err != nil {
            return err
        }
    }
    return nil
}
```

Эта функция создает пустой файл zip-архива, потом создает значение типа `*zip.Writer` (`zipper`) для записи в файл и затем выполняет итерации по всем файлам, записывая их по очереди в файл архива.

Обе функции, `createZip()` и `writeFileToZip()`, очень короткие, и поэтому может появиться соблазн объединить их в одну функцию. Однако это было бы неблагоразумно, потому что в этом случае в цикле `for` придется открывать файл за файлом (то есть в конечном итоге все файлы в срезе `files`), в результате чего можно исчерпать предел операционной системы на количество одновременно открытых файлов, о чем коротко упоминалось в предыдущей главе (§7.2.5). Конечно, вместо того чтобы откладывать вызов метода `os.File.Close()`, его можно было бы выполнять в каждой итерации, но тогда пришлось бы каким-то образом гарантировать закрытие файла независимо от того, произошла ошибка или нет. Поэтому для работы с отдельными файлами проще всегда создавать отдельные функции, как это сделано здесь.

```
func writeFileToZip(zipper *zip.Writer, filename string) error {
    file, err := os.Open(filename)
    if err != nil {
        return err
    }
}
```

```
defer file.Close()
info, err := file.Stat()
if err != nil {
    return err
}
header, err := zip.FileInfoHeader(info)
if err != nil {
    return err
}
header.Name = sanitizedName(filename)
writer, err := zipper.CreateHeader(header)
if err != nil {
    return err
}
_, err = io.Copy(writer, file)
return err
}
```

Функция сначала открывает на чтение файла для сжатия и откладывает его закрытие уже знакомым способом.

Далее вызывается метод `os.File.Stat()`, извлекающий время последнего изменения файла и биты разрешений на доступ к нему в виде значения `os.FileInfo`. Это значение передается функции `zip.FileInfoHeader()`, возвращающей значение типа `zip.FileHeader` с заполненным временем последнего изменения, битами разрешений и именем файла. Программа необязательно должна сохранять в архиве файл с его оригинальным именем, и здесь имя файла (в поле `zip.FileHeader.Name`) замещается его откорректированной версией.

После настройки заголовка вызывается функция `zip.CreateHeader()`, которой в качестве аргумента передается подготовленный заголовок. Она создает запись в zip-архиве, используя время, разрешения и имя из заголовка, и возвращает значение типа `io.Writer`, которое можно использовать для записи содержимого файла в архив. Для этой цели здесь используется функция `io.Copy()` — она возвращает число скопированных байт (которое здесь просто отбрасывается) и либо `nil`, либо признак ошибки.

В случае ошибки она немедленно возвращается вызывающей программе для обработки. А в случае успеха в конце указанный файл будет сохранен в архиве.

```
func sanitizedName(filename string) string {
    if len(filename) > 1 && filename[1] == ':' &&
        runtime.GOOS == "windows" {
        filename = filename[2:]
    }
    filename = filepath.ToSlash(filename)
    filename = strings.TrimLeft(filename, "/.")
    return strings.Replace(filename, "../", "", -1)
}
```

Если сохранять в архиве имена файлов с абсолютными путями или путями, содержащими компонент `..`, при распаковывании такого архива можно непреднамеренно затереть важные файлы. Чтобы уменьшить риск, программа корректирует имена файлов, сохраняемых в архиве.

Функция `sanitizedName()` убирает из пути к файлу букву диска и двоеточие (если они указаны), затем удаляет ведущие символы-разделители каталогов и точки и все компоненты `..` пути. Кроме того, она принудительно замещает все символы-разделители элементов пути на символы прямого слеша.

8.2.2. Создание тарболлов

Создание *тарболлов* реализуется почти так же, как создание zip-архивов, с важным отличием, которое заключается в том, что все данные записываются в одно и то же значение, предназначенное для записи, и перед содержимым файла необходимо записывать полный заголовок, а не только имя файла. Реализация поддержки тарболлов в программе `pack` состоит из двух функций, `createTar()` и `writeFileToTar()`.

```
func createTar(filename string, files []string) error {
    file, err := os.Create(filename)
    if err != nil {
        return err
    }
    defer file.Close()
    var fileWriter io.WriteCloser = file
    if strings.HasSuffix(filename, ".gz") {
        fileWriter = gzip.NewWriter(file)
        defer fileWriter.Close()
    }
}
```

```

writer := tar.NewWriter(fileWriter)
defer writer.Close()
for _, name := range files {
    if err := writeFileToTar(writer, name); err != nil {
        return err
    }
}
return nil
}

```

Эта функция создает файл тарболла с указанным именем и добавляет фильтр gzip-сжатия, если расширение в имени файла указывает на необходимость сжатия тарболла. Функция `gzip.NewWriter()` возвращает значение типа `*gzip.Writer`, реализующего интерфейс `io.WriteCloser` (подобно значению `*os.File`, соответствующему открытому файлу).

После подготовки файла функция создает значение типа `*tar.Writer` для записи в него. Затем она выполняет итерации по всем файлам и пытается записать их по очереди.

```

func writeFileToTar(writer *tar.Writer, filename string) error {
    file, err := os.Open(filename)
    if err != nil {
        return err
    }
    defer file.Close()
    stat, err := file.Stat()
    if err != nil {
        return err
    }
    header := &tar.Header{
        Name:    sanitizedFilename(filename),
        Mode:    int64(stat.Mode()),
        Uid:     os.Getuid(),
        Gid:     os.Getgid(),
        Size:    stat.Size(),
        ModTime: stat.ModTime(),
    }
    if err = writer.WriteHeader(header); err != nil {
        return err
    }
    _, err = io.Copy(writer, file)
    return err
}

```

Эта функция сначала открывает указанный файл для чтения и откладывает его закрытие. Затем выполняется системный вызов *stat* для файла, чтобы получить флаги режимов файла, размер и дату/время последнего его изменения – эти данные используются для заполнения значения типа **tar.Header*, которое должно создаваться для каждого файла, сохраняемого в тарболле. (Дополнительно в заголовке сохраняются числовые идентификаторы пользователя и группы – они используются в Unix-подобных системах.) Для каждого файла в заголовке должны указываться хотя бы имя файла (поле *Name*) и размер (поле *Size*), иначе будет создан недопустимый тарболл.

После создания и заполнения заголовка типа **tar.Header* он записывается в тарболл. И затем в тарболл записывается содержимое файла.

8.2.3. Распаковывание zip-архивов

Распаковывание zip-архивов выполняется так же просто, как и упаковывание, единственное только нужно – воссоздать структуру каталогов, если имена файлов в архиве содержат пути.

```
func unpackZip(filename string) error {
    reader, err := zip.OpenReader(filename)
    if err != nil {
        return err
    }
    defer reader.Close()
    for _, zipFile := range reader.Reader.File {
        name := sanitizedFilename(zipFile.Name)
        mode := zipFile.Mode()
        if mode.IsDir() {
            if err = os.MkdirAll(name, 0755); err != nil {
                return err
            }
        } else {
            if err = unpackZippedFile(name, zipFile); err != nil {
                return err
            }
        }
    }
    return nil
}
```

Эта функция открывает указанный zip-архив для чтения. Вместо последовательности вызовов функций `os.Open()` и `zip.NewReader()` пакет `zip` позволяет вызвать единственную функцию `zip.OpenReader()`, которая выполнит обе операции и вернет значение типа `*zip.ReadCloser` для работы с архивом. Наиболее важной особенностью значения `zip.ReadCloser` является его экспортируемое поле составного типа `zip.Reader`, содержащее значение типа `[]*zip.File` – срез с указателями на структуры `zip.File`, каждая из которых представляет файл в zip-архиве.

Функция выполняет итерации по структурам `zip.File` и создает скорректированные имена файлов и каталогов (используя ту же функцию `sanitizedName()`, что применяется в программе `pack`), чтобы уменьшить риск затирания важных файлов.

Если очередной элемент в архиве является каталогом (как сообщает метод `IsDir()`), функция пытается создать этот каталог. Функция `os.MkdirAll()` обладает замечательным свойством создавать все промежуточные каталоги при создании конкретного каталога. Если требуемый каталог уже существует, она ничего не делает и просто возвращает `nil`¹. Если очередной элемент является файлом, его распаковывание передается функции `unpackZippedFile()`.

```
func unpackZippedFile(filename string, zipFile *zip.File) error {
    writer, err := os.Create(filename)
    if err != nil {
        return err
    }
    defer writer.Close()
    reader, err := zipFile.Open()
    if err != nil {
        return err
    }
    defer reader.Close()
    if _, err = io.Copy(writer, reader); err != nil {
        return err
    }
    if filename == zipFile.Name {
        fmt.Println(filename)
    }
}
```

¹ Как отмечалось выше, флаги разрешений для файла традиционно записываются в виде восьмеричных чисел, где типичным значением для файлов является число 0666. Для каталогов типичным значением является число 0755.

```
    } else {  
        fmt.Printf("%s [%s]\n", filename, zipFile.Name)  
    }  
    return nil  
}
```

Эта функция извлекает единственный файл из архива и сохраняет его в файловой системе. Как обычно, она начинается с создания файла для записи. Затем она открывает указанный архив с помощью функции `zip.File.Open()` и записывает содержимое файла в архиве во вновь созданный файл.

В конце, если не возникло никаких ошибок, функция выводит в консоль имя созданного файла, указывая оригинальное имя в квадратных скобках, если скорректированное имя отличается от него.

Тип `*zip.File` имеет еще ряд полезных методов, таких как `zip.File.Mode()` (используется в функции `unpackZip()` выше), `zip.File.ModTime()` (возвращает время последнего изменения файла в виде значения типа `time.Time`) и `zip.FileInfo()` (возвращает значение типа `os.FileInfo` для файла).

8.2.4. Распаковывание тарболлов

Распаковывание tar-файлов выполняется немного проще, чем их создание. Однако, как и в случае распаковывания zip-архивов, необходимо предусмотреть воссоздание структуры каталогов, если имена файлов в архиве включают пути к ним.

```
func unpackTar(filename string) error {  
    file, err := os.Open(filename)  
    if err != nil {  
        return err  
    }  
    defer file.Close()  
    var fileReader io.ReadCloser = file  
    if strings.HasSuffix(filename, ".gz") {  
        if fileReader, err = gzip.NewReader(file); err != nil {  
            return err  
        }  
        defer fileReader.Close()  
    }  
    reader := tar.NewReader(fileReader)  
    return unpackTarFiles(reader)  
}
```

Эта функция открывает тарболл обычным для языка Go способом и откладывает его закрытие. Если для файла использовалось gzip-сжатие, добавляется фильтр gzip-декомпрессии и откладывается его закрытие. Функция `gzip.NewReader()` возвращает значение типа `*gzip.Reader`, реализующего интерфейс `io.ReadCloser`, подобно файлам (типа `*os.File`).

После настройки значения для чтения файла создается значение типа `*tar.Reader` для чтения из архива, и вся остальная работа перекладывается на вспомогательную функцию.

```
func unpackTarFiles(reader *tar.Reader) error {
    for {
        header, err := reader.Next()
        if err != nil {
            if err == io.EOF {
                return nil // OK
            }
            return err
        }
        filename := sanitizedHeaderName(header.Name)
        switch header.Typeflag {
        case tar.TypeDir:
            if err = os.MkdirAll(filename, 0755); err != nil {
                return err
            }
        case tar.TypeReg:
            if err = unpackTarFile(filename, header.Name, reader);
                err != nil {
                return err
            }
        }
    }
    return nil
}
```

Эта функция выполняет бесконечный цикл, производя итерации по всем записям в тарболле, пока не получит признака `io.EOF` (или пока не возникнет ошибка). Метод `tar.Reader.Next()` возвращает первую или следующую запись из тарболла в виде значения типа `*tar.Header` или сообщает об ошибке. Если была получена ошибка со значением `io.EOF`, это сообщает, что достигнут конец архива, поэтому функция возвращает пустое значение ошибки.

Если значение типа `*tar.Header` было получено успешно, на основе поля `Name` в заголовке создается скорректированное имя файла. Далее следует инструкция `switch`, производящая выбор в зависимости от типа записи. В этом простом примере обрабатываются только обычные файлы и каталоги, но в действительности тарболлы могут содержать записи других типов (например, символические ссылки).

Если запись соответствует каталогу, создается каталог, точно так же, как это делалось в реализации распаковывания `zip`-архивов. А если запись соответствует файлу, работа по его распаковыванию передается вспомогательной функции.

```
func unpackTarFile(filename, tarFilename string,
    reader *tar.Reader) error {
    writer, err := os.Create(filename)
    if err != nil {
        return err
    }
    defer writer.Close()
    if _, err = io.Copy(writer, reader); err != nil {
        return err
    }
    if filename == tarFilename {
        fmt.Println(filename)
    } else {
        fmt.Printf("%s [%s]\n", filename, tarFilename)
    }
    return nil
}
```

Эта функция создает новый файл, соответствующий следующей записи в тарболле, и откладывает его закрытие. Затем она копирует данные из записи в файл. И точно так же, как функция `unpackZippedFile()`, она выводит в консоль имя созданного файла, с оригинальным именем файла в квадратных скобках, если оно отличается от скорректированного имени.

На этом завершается обзор сжатых и архивных файлов и обработки файлов в целом. Подход к работе с файлами в языке Go основан на использовании интерфейсов `io.Reader`, `io.ReadCloser`, `io.Writer` и `io.WriteCloser`, упрощающих выполнение операций чтения и записи файлов или других потоков (таких как сетевые соединения и даже строки) с использованием совершенно одинаковых приемов.

8.3. Упражнения

В этой главе предлагается выполнить три упражнения. Первое хоть и небольшое, но связано с очень тонким изменением одной из программ, представленных в этой главе. Во втором потребуются написать короткую, но достаточно сложную новую программу с нуля. Третье связано с внесением существенных изменений в другой пример из этой главы.

1. Скопируйте содержимое каталога `unpack`, например, в каталог `my_unpack` и измените программу `unpack.go` так, чтобы она могла дополнительно распаковывать архивы `.tar.bz2` (с `bzip2`-сжатием). Для этого придется внести небольшие изменения в пару функций и добавить около десятка строк в функцию `unpackTar()`. Здесь придется немного подумать, потому что функция `bzip2.NewReader()` не возвращает значения типа `io.ReadCloser`. Одно из возможных решений представлено в файле `unpack_ans/unpack.go`, который примерно на десять строк длиннее файла из оригинального примера.
2. Для текстовых файлов в Windows (`.txt`) часто используется кодировка UTF-16-LE (UTF-16 little-endian (обратный порядок следования байтов)). Файлы в кодировке UTF-16 всегда должны начинаться с маркера, определяющего порядок следования байтов, `[0xFF, 0xFE]` – для обратного порядка и `[0xFE, 0xFF]` – для прямого. Напишите программу, которая будет читать файлы в кодировке UTF-16, принимая их имена в виде аргументов командной строки, и выводить тот же текст в кодировке UTF-8 либо в `os.Stdout`, либо в файл, имя которого также может быть указано в аргументе командной строки. Убедитесь, что программа корректно обрабатывает файлы в обеих разновидностях кодировки UTF-16, с прямым и обратным порядком следования байтов. В составе загружаемых примеров к книге имеется пара небольших тестовых файлов: `utf16-to-utf8/utf-16-be.txt` и `utf16-to-utf8/utf-16-le.txt`. В пакете `binary` имеется функция `Read()`, которая может читать значения типа `uint16` (каковыми являются символы в кодировке UTF-16), учитывая указанный порядок следования байтов. А в пакете `unicode/utf16` имеется функция `Decode()`, преобразующая срез со значениями типа `uint16` в срез с кодовыми пунктами (то есть в срез типа `[]rune`), благодаря чему достаточно будет обернуть результат вызова функции `utf16.Decode()`

операцией приведения типа `string()`, чтобы создать строку в кодировке UTF-8. Одно из возможных решений представлено в файле `utf16-to-utf8/utf16-to-utf8.go` и занимает около 50 строк, исключая раздел импорта.

3. Скопируйте содержимое каталога `invoicedata`, например, в каталог `my_invoicedata` и внесите в программу `invoicedata` несколько изменений. Во-первых, измените определения структур `Invoice` и `Item`, как показано ниже.

```

type Invoice struct { // версия файла
    Id          int        // 100
    CustomerId  int        // 100
    DepartmentId string    // 101
    Raised      time.Time  // 100
    Due         time.Time  // 100
    Paid        bool       // 100
    Note        string     // 100
    Items       []*Item    // 100
}

type Item struct { // версия файла
    Id      string // 100
    Price   float64 // 100
    Quantity int    // 100
    TaxBand int    // 101
    Note    string // 100
}

```

Теперь измените программу так, чтобы она всегда сохраняла накладные в новом формате (то есть обрабатывала бы новые структуры) и могла читать файлы и в оригинальном, и в новом формате.

При чтении данных в оригинальном формате нельзя использовать нулевые значения для новых полей – эти поля должны заполняться в соответствии со следующими правилами: в поле `DepartmentId` должна записываться строка "GEN", если поле `InvoiceID` имеет значение меньше 3000; "МКТ" – если меньше 4000, "COM" – если меньше 5000, "EXP" – если меньше 6000, "INP" – если меньше 7000, "TZZ" – если меньше 8000, "V20" – если меньше 9000, и "X15" во всех остальных случаях. В поле `TaxBand` структуры `Item` должно быть записано целое число, равное значению третьего символа в поле `Id` этой же структуры. Например, если в поле `Id` хранится строка "JU4661", в поле `TaxBand` следует записать число 4.

Одно из возможных решений представлено в каталоге `invoicedata_ans`. В предложенном решении добавлены три функции (в файле `invoicedata.go`): одна обновляет все накладные в срезе типа `[]*Invoice` (то есть подставляет приемлемые значения в новые поля), одна обновляет одну накладную, и

одна обновляет один пункт накладной. Чтобы выполнить это упражнение, потребуется изменить все файлы `.go`: больше всего изменений придется внести в файлы `jsn.go`, `xml.go` и `txt.go`. В сумме изменения занимают около 150 строк дополнительного программного кода.



9. Пакеты

Стандартная библиотека Go включает огромное количество пакетов, предоставляющих самые разные функциональные возможности. Вдобавок к этому на сайте Go Dashboard (godashboard.appspot.com/project) можно найти массу сторонних пакетов.

Кроме того, язык Go позволяет пользователям создавать собственные пакеты. Эти пакеты можно установить в свою копию стандартной библиотеки или хранить в отдельных каталогах (пути к которым должны быть включены в переменную окружения `GOPATH`).

В этой главе рассказывается, как создавать и импортировать *пакеты*, включая пользовательские и сторонние пакеты. Затем очень коротко будут представлены некоторые команды компилятора `gc`. И в заключение будет дан небольшой обзор стандартной библиотеки Go, чтобы можно было избежать реализации функциональных возможностей, уже имеющихся в библиотеке.

9.1. Пользовательские пакеты

До сих пор практически все примеры в книге состояли из единственного пакета `main`. Программный код любого пакета можно разбить на множество файлов, при условии что все они будут находиться в одном каталоге. Например, в главе 8 в примере `invoicedata` используется единственный пакет (`main`), состоящий из шести отдельных файлов (`invoicedata.go`, `gob.go`, `inv.go`, `jsn.go`, `txt.go` и `xml.go`). Это достигается за счет того, что первой инструкцией (кроме комментариев) в каждом файле является инструкция `package main`.

При разработке больших приложений может потребоваться создать пакеты специально для данного приложения, чтобы разбить функциональные возможности на логические модули. Также может потребоваться создать пакеты с функциональными возможностями для целого семейства приложений. Пакеты, предназначенные для использования в единственном приложении и во множестве приложений, в языке Go никак не различаются. Однако такое различие

можно симитировать, поместив пакеты, предназначенные для конкретного приложения, в подкаталоги самого приложения, а совместно используемые пакеты – в подкаталоги с *исходными текстами*, находящиеся непосредственно в каталогах, перечисленных в переменной окружения GOPATH. Под *каталогом с исходными текстами* в пути GOPATH подразумевается каталог с именем src – каждый каталог, перечисленный в переменной окружения GOPATH, должен содержать подкаталог src, потому что именно на это имя ориентируются инструменты (команды) компилятора Go. Исходный программный код приложений и пакетов должен храниться в подкаталогах, находящихся в каталогах src в пути GOPATH.

Пользовательские пакеты можно также установить непосредственно в дерево Go (то есть в каталог GOROOT), но это не дает никаких преимуществ и может оказаться неудобным в системах, где Go установлен с помощью системы управления пакетами, мастера установки или даже если компилятор был собран вручную.

9.1.1. Создание пользовательских пакетов

Создавать собственные пакеты лучше всего в одном из каталогов src в пути GOPATH. Пакеты, предназначенные для конкретного приложения, можно создавать в каталоге приложения, но пакеты для использования в нескольких приложениях следует создавать непосредственно в каталогах src в пути GOPATH, в идеале – в отдельном каталоге, чтобы избежать конфликтов имен.

В соответствии с соглашениями исходный программный код пакета должен сохраняться в каталоге с тем же именем, что и имя пакета. Исходный код можно разместить в нескольких файлах, и файлы могут иметь произвольные имена (при условии что они имеют расширение .go). Примеры в этой книге следуют соглашению, в соответствии с которым файлам с расширением .go (или одному из них, если их несколько) даются имена, совпадающие с именами пакетов.

Пример stacker из главы 1 (§1.5), состоящий из программы (в файле stacker.go) и пакета stack (в файле stack.go), имеет следующую организацию каталогов:

aGoPath/src/stacker/stacker.go

aGoPath/src/stacker/stack/stack.go

Здесь под именем aGoPath подразумевается один из каталогов, перечисленных в переменной окружения GOPATH.

Если перейти в каталог `stacker` и выполнить команду `go build`, будет создан выполняемый файл с именем `stacker` (`stacker.exe` – в Windows). Но чтобы поместить выполняемый файл в каталог `bin` в пути `GOPATH` или чтобы сделать пакет `stacker/stack` доступным для использования в других программах, необходимо выполнить команду `go install`.

Когда команда `go install` соберет программу `stacker`, она создаст два каталога (если они отсутствуют): `aGoPath/bin` с выполняемым файлом `stacker` и `aGoPath/pkg/linux_amd64/stacker` с файлом статической библиотеки пакета `stack`. (Естественно, имя каталога, отражающего тип операционной системы и аппаратную архитектуру, будет соответствовать компьютеру, на котором выполняется сборка, например в 32-битной версии Windows каталог получит имя `windows_386`.)

Пакет `stack` может быть импортирован программой `stacker` с помощью инструкции `import "stacker/stack"`, то есть с указанием полного пути (в стиле Unix), но исключая часть пути `aGoPath/src`. В действительности *любая* программа или пакет в пути `GOPATH` смогут использовать эту инструкцию импортирования, потому что в языке Go нет различий между пакетами, созданными для конкретного приложения, и общими пакетами.

Пример реализации упорядоченного отображения из главы 6 (§6.5.3; в файле `omap.go`) является пакетом `omap`. Этот пакет предназначен для использования множеством приложений. Чтобы избежать конфликтов имен, необходимо создать в одном из каталогов в пути `GOPATH` каталог для пакетов и дать ему уникальное имя (в данном случае каталогу было присвоено имя домена). Ниже приводится структура каталога:

`aGoPath/src/qtrac.eu/omap/omap.go`

Любая разрабатываемая программа (при условии, что она находится в пути `GOPATH`) сможет импортировать пакет с реализацией упорядоченного отображения, выполнив инструкцию `import "qtrac.eu/omap"`. Создавая другой пакет, его можно было бы поместить в каталог `aGoPath/src/qtrac.eu` рядом с пакетом упорядоченного отображения.

Когда команда `go install` соберет пакет `omap`, она создаст каталог `aGoPath/pkg/linux_amd64/qtrac.eu` (если он отсутствует) с файлом библиотеки пакета `omap` и с подкаталогом, имя которого соответствует операционной системе и аппаратной архитектуре.

Если потребуется создать пакет внутри другого пакета, для этого не придется делать что-то особенное. Сначала нужно будет создать каталог пакета, например `aGoPath/src/my_package`. А затем для каждого вложенного пакета создать отдельный подкаталог, например `aGoPath/src/my_package/pkg1` и `aGoPath/src/my_package/pkg2`, а также соответствующие им файлы `aGoPath/src/my_package/pkg1/pkg1.go` и `aGoPath/src/my_package/pkg2/pkg2.go`. После этого, чтобы импортировать, допустим, пакет `pkg2`, можно будет использовать инструкцию `import "my_package/pkg2"`. Такой подход, например, используется для организации программного кода пакета `archive`. В самом пакете также можно разместить программный код (для данного примера, создав файл `aGoPath/src/my_package/my_package.go`). В качестве примера применения такого подхода можно привести пакет `image`.

Импортируемые пакеты должны находиться в каталоге `GOROOT` (в частности, `$GOROOT/pkg/${GOOS}_${GOARCH}`, например `/opt/go/pkg/linux_amd64`) или в одном из каталогов, перечисленных в переменной окружения `GOPATH`. Это означает возможность конфликтов имен. Проще всего избежать их, создавая в пути `GOPATH` каталоги с уникальными именами, такими как имена доменов, как это было сделано при создании пакета `otap`.

Программа `go` без каких-либо проблем работает с пакетами из стандартной библиотеки и с пакетами из пути `GOPATH`, а также различает файлы с исходным программным кодом, зависящим от платформы, которые обсуждаются ниже.

9.1.1.1. Программный код, зависящий от платформы

В некоторых ситуациях бывает необходимо иметь программный код, разный для разных платформ. Например, в Unix-подобных системах командная оболочка поддерживает *подстановку для шаблонных символов*, то есть последовательность символов `*.txt` в командной строке может быть принята программой как, например, `["README.txt", "INSTALL.txt"]` в срезе `os.Args[1:]`. Но в Windows программа просто получит `["*.txt"]`. Сымитировать такую подстановку для шаблонных символов можно непосредственно в самой программе с помощью функции `filepath.Glob()`, но это необходимо только в Windows.

Решение об использовании функции `filepath.Glob()` может быть принято во время выполнения с помощью проверки `if runtime.GOOS == "windows" {...}`, и именно этот способ используется в большинстве примеров из книги (например, `cgrep1/cgrep.go`). Однако *платформенно зависимый программный код* можно поместить в

отдельный файл или файлы с расширением `.go`. Например, программа `cgrep3` состоит из трех файлов: `cgrep.go`, `util_linux.go` и `util_windows.go`. В файле `util_linux.go` определена всего одна функция.

```
func commandLineFiles(files []string) []string { return files }
```

Очевидно, что эта функция ничего не делает для подстановки шаблонных символов, потому что в Linux в этом нет необходимости. В файле `util_windows.go` определена одноименная функция, отличающаяся своей реализацией.

```
func commandLineFiles(files []string) []string {
    args := make([]string, 0, len(files))
    for _, name := range files {
        if matches, err := filepath.Glob(name); err != nil {
            args = append(args, name) // Недопустимый шаблон
        } else if matches != nil {    // Хотя бы одно совпадение
            args = append(args, matches...)
        }
    }
    return args
}
```

При сборке программы `cgrep3` командой `go build` в Linux будет компилироваться файл `util_linux.go`, а файл `util_windows.go` будет проигнорирован. В Windows все будет происходить наоборот. Этим гарантируется, что скомпилирована будет только одна из функций `commandLineFiles()`.

В Mac OS X и FreeBSD ни один из файлов, `util_linux.go` и `util_windows.go`, не будет скомпилирован, что приведет к ошибке сборки. Поскольку командные оболочки в обеих этих платформах выполняют подстановку, можно либо создать символические ссылки, либо скопировать содержимое `util_linux.go` в файлы `util_darwin.go` и `util_freebsd.go` (и аналогично для любых других платформ, поддерживаемых языком Go и где это необходимо). При наличии упомянутых символических ссылок или копий программа будет собираться в Mac OS X и FreeBSD.

9.1.1.2. Документирование пакетов

Пакеты, особенно те, что предназначены для совместного использования, следует тщательно *документировать*. В комплект языка Go входит инструмент для создания документации, `godoc`,

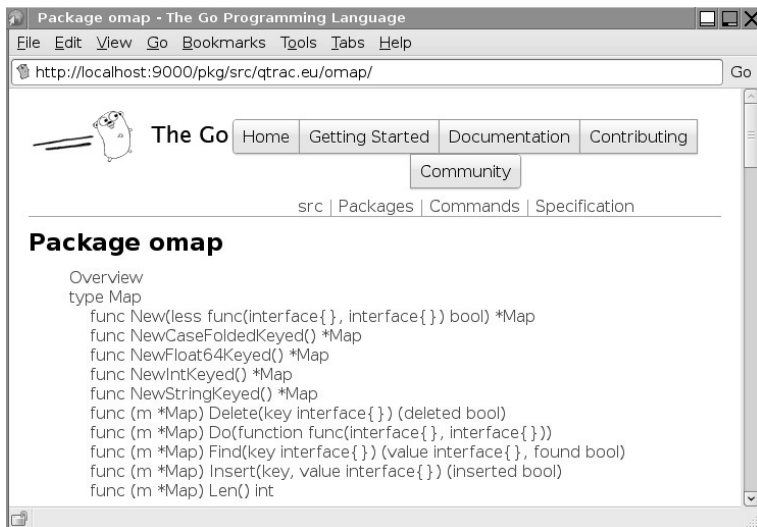


Рис. 9.1. Документация для пакета *omap*

который можно использовать для отображения документации с описанием пакетов и функций в консоли или в браузере, в виде веб-страниц, как показано на рис. 9.1¹. Если пакет находится в пути GOPATH, программа `godoc` автоматически найдет его и создаст ссылку левее ссылки **Packages** (Пакеты). Если пакет находится не в пути GOPATH, запустите программу `godoc` с параметром `-path` (в дополнение к параметру `-http`), указав в нем путь к пакету, и `godoc` снова выведет ссылку рядом со ссылкой **Packages** (Пакеты). (Инструмент `godoc` обсуждался во врезке «Документация по языку Go», в главе 1.)

Что должно быть в хорошей документации – спорный вопрос, поэтому здесь мы сосредоточимся исключительно на механике документирования пакетов.

По умолчанию `godoc` отображает экспортируемые типы, классы, константы и переменные, поэтому все эти элементы должны документироваться. Документация помещается непосредственно в файлы с исходными текстами.

¹ На снимках с экрана демонстрируется, как выглядели HTML-страницы, создаваемые инструментом `godoc`, на момент написания этих строк. Они могут измениться к тому моменту, как книга окажется у вас в руках.

```
// Пакет omap реализует эффективное отображение, упорядоченное по ключам.
//
// Ключи и значения могут быть любого типа, но ключи должны поддерживать операцию
// сравнения с помощью функции "меньше чем", которая должнв передаваться
// функции omap.New(), или функций по умолчанию, предоставляемых
// функцией-конструктором omap.New*().
package omap
```

Комментарий, непосредственно предшествующий инструкции `package`, интерпретируется как описание пакета, где первая строка (до первой точки, если таковая имеется, или до первого символа перевода строки) играет роль краткого однострочного описания. Этот пример взят из пакета `omap` (в файле `qtrac.eu/omap/omap.go` — этот пакет рассматривался в главе 6, в §6.5.3).

```
// Map — это упорядоченное отображение.
// Нулевое значение этого типа является недопустимым отображением! Для создания
// отображения с ключами определенного типа используйте одну из функций-конструкторов
// (например, New()).
type Map struct {
```

Описание экспортируемых типов должно помещаться непосредственно перед инструкцией `type` и всегда должно подчеркивать допустимость нулевых значений этих типов.

```
// New возвращает пустое отображение типа Map, использующее указанную
// функцию "меньше чем" для сравнения ключей. Например:
//     type Point { X, Y int }
//     pointMap := omap.New(func(a, b interface{}) bool {
//         α, β := a.(Point), b.(Point)
//         if α.X != β.X {
//             return α.X < β.X
//         }
//         return α.Y < β.Y
//     })
func New(less func(interface{}, interface{}) bool) *Map {
```

Описание функций и методов должно непосредственно предшествовать их первой строке. Здесь приводится описание обобщенной функции-конструктора `New()` в пакете `omap`.

На рис. 9.2 показано, как выглядит это описание функции в веб-странице, сгенерированной инструментом `godoc`. На данном рисунке также видно, что текст с отступами отображается как «программный код». Однако на момент написания этих строк `godoc` не поддерживал

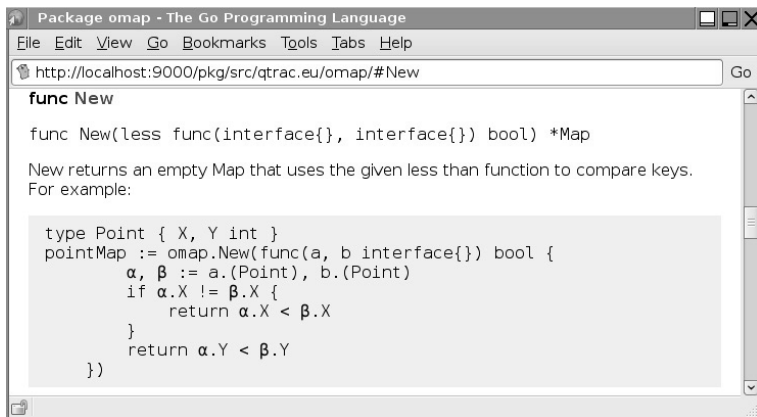


Рис. 9.2. Документация с описанием функции `New()` в пакете `omap`

какую бы то ни было разметку (например, возможность явно указать использование жирного или курсивного шрифта, создать ссылку и т. д.).

```
// NewCaseFoldedKeyed возвращает пустое отображение Map со строковыми
// ключами, нечувствительными к регистру символов.
func NewCaseFoldedKeyed() *Map {
```

Фрагмент выше демонстрирует описание одной из удобных функций-конструкторов, реализующей предопределенную функцию сравнения «меньше чем».

```
// Insert вставляет в отображение Map новую пару ключ/значение и возвращает
// true; или замещает имеющееся значение, если указанный ключ уже имеется в
// отображении, и возвращает returns false. Например:
// inserted := myMap.Insert(key, value).
func (m *Map) Insert(key, value interface{}) (inserted bool) {
```

А здесь приводится описание метода `Insert()`. Обратите внимание, что в соответствии с соглашениями, принятыми в языке Go, описание функций и методов начинается с их имен, и (вопреки традиции) круглые скобки после имен не указываются.

9.1.1.3. Модульное тестирование и измерение эффективности пакетов

Стандартная библиотека Go включает отличную поддержку для проведения модульного тестирования с помощью пакета `testing`.

Подготовка к *модульному тестированию* пакета заключается в создании тестового файла в каталоге этого пакета. Имя этого файла должно начинаться с имени пакета и заканчиваться `_test.go`. Например, тестовый файл для пакета `omap` называется `omap_test.go`.

В загружаемых примерах к книге тестовые файлы помещены в отдельные пакеты (например, `omap_test`) и импортируют тестируемые пакеты, а также пакет `testing` плюс и другие необходимые пакеты. Это вынуждает проводить тестирование по принципу «черного ящика». Однако некоторые программисты предпочитают использовать тестирование по принципу «стеклянного ящика». Этого легко можно добиться, поместив тестовые файлы в каталоги тестируемых ими пакетов (например, `omap`) – в этом случае отпадает необходимость импортировать тестируемые пакеты. Последний подход позволяет тестировать неэкспортируемые типы и добавлять методы в неэкспортируемые типы для нужд тестирования.

Тестовые файлы необычны тем, что не содержат функцию `main()`. Вместо этого они содержат одну или более экспортируемых функций, имена которых начинаются с `Test`, принимающих единственный аргумент типа `*testing.T` и ничего не возвращающих. Конечно, в тестовых файлах можно также определять любые вспомогательные функции, имена которых не начинаются с `Test`.

```
func TestStringKeyOMapInsertion(t *testing.T) {
    wordForWord := omap.NewCaseFoldedKeyed()
    for _, word := range []string{"one", "Two", "THREE", "four", "Five"} {
        wordForWord.Insert(word, word)
    }
    var words []string
    wordForWord.Do(func(_, value interface{}) {
        words = append(words, value.(string))
    })
    actual, expected := strings.Join(words, ""), "FivefouroneTHREETwo"
    if actual != expected {
        t.Errorf("%q != %q", actual, expected)
    }
}
```

Это один из тестов в файле `omap_test.go`. Он начинается с создания пустого отображения типа `omap.Map`, затем вставляет несколько строковых ключей (которые интерпретируются без учета регистра символов) и строковых значений. Потом он выполняет итерации по всем

парам *ключ/значение* с помощью метода `Map.Do()` и добавляет каждое значение в срез со строками. Наконец он объединяет все строки в одну строку и сравнивает ее с ожидаемым результатом. В случае несоответствия строк вызывается метод `testing.T.Errorf()`, чтобы сообщить об ошибке и дать некоторые пояснения. Если в процессе выполнения ошибок не произошло, тест считается пройденным.

Ниже приводится пример тестирования, где тесты проходятся без ошибок.

```
$ go test
```

```
ok qtrac.eu/omap
```

```
PASS
```

А вот попытка выполнить то же самое тестирование, но на этот раз с ключом `-test.v`, включающим вывод подробного отчета о тестировании.

```
$ go test -test.v
```

```
ok      qtrac.eu/omap
```

```
=== RUN TestStringKey0MapInsertion-4
```

```
--- PASS: TestStringKey0MapInsertion-4 (0.00 seconds)
```

```
=== RUN TestIntKey0MapFind-4
```

```
--- PASS: TestIntKey0MapFind-4 (0.00 seconds)
```

```
=== RUN TestIntKey0MapDelete-4
```

```
--- PASS: TestIntKey0MapDelete-4 (0.00 seconds)
```

```
=== RUN TestPassing-4
```

```
--- PASS: TestPassing-4 (0.00 seconds)
```

```
PASS
```

Если изменить литерал строки, чтобы тестирование завершалось ошибкой, и снова запустить его (в кратком режиме по умолчанию), будет получен следующий вывод.

```
$ go test
```

```
FAIL      qtrac.eu/omap
```

```
--- FAIL: TestStringKey0MapInsertion-4 (0.01 seconds)
```

```
omap_test.go:35: "FivefouroneTHREETwo" != "FivefouroneTHREEToo"
```

```
FAIL
```

Помимо метода `Errorf()`, использованного в этом примере, тип `*testing.T` из пакета `testing` имеет множество других методов, таких как `testing.T.Fail()`, `testing.T.Fatal()`, и т. д. Все эти методы

предоставляют достаточно полный контроль над тем, что должно выводиться в случае ошибок.

Кроме всего перечисленного выше, пакет `testing` поддерживает возможность *измерения эффективности*. Функции хронометража можно добавлять в файлы `имя_пакета_test.go`, так же как и функции тестирования, только эти функции должны иметь имена, начинающиеся со слова `Benchmark`, принимать единственный аргумент типа `*testing.B` и ничего не возвращать.

```
func BenchmarkOMapFindSuccess(b *testing.B) {
    b.StopTimer() // Не учитывать время на создание и заполнение
    intMap := omap.NewIntKeyed()
    for i := 0; i < 1e6; i++ {
        intMap.Insert(i, i)
    }
    b.StartTimer() // Выполнить замер эффективности успешного поиска методом
Find()
    for i := 0; i < b.N; i++ {
        intMap.Find(i % 1e6)
    }
}
```

Эта функция сначала останавливает таймер, чтобы исключить из замера время, необходимое на создание и заполнение отображения типа `omap.Map`. Затем она создает новое отображение и заполняет его миллионом пар *ключ/значение*.

По умолчанию команда `go test` не проводит хронометраж, поэтому о его необходимости требуется сообщать явно, передавая параметр `-test.bench` с регулярным выражением для выбора необходимых функций хронометража. Регулярному выражению `.*` соответствуют любые имена, то есть все функции хронометража в тестовом файле, однако простой точки `.` также будет достаточно.

```
$ go test -test.bench=.
```

```
PASS      qtrac.eu/omap
```

```
PASS
```

```
BenchmarkOMapFindSuccess-4      1000000      1380 ns/op
```

```
BenchmarkOMapFindFailure-4      1000000      1350 ns/op
```

Данный вывод показывает, что было выполнено два измерения производительности, в каждом из которых выполнено по 1 000 000 итераций, и на каждую итерацию было затрачено указанное количество наносекунд. Количество итераций (то есть значение `b.N`)

выбирается командой `go test`, однако при желании можно воспользоваться параметром `-test.benchtime`, чтобы указать примерное количество секунд на каждое измерение.

Кроме пакета `os`, в загружаемых примерах к книге имеется еще несколько примеров файлов `имя_пакета_test.go`.

9.1.2. Импортирование пакетов

В языке Go имеется возможность импортировать пакеты под *псевдонимами*. Это может пригодиться, например, чтобы упростить переход от одной версии пакета к другой. Например, программа может импортировать пакет как `import bio "bio_v1"`, чтобы пакет `bio_v1` был доступен программному коду под именем `bio`. Позднее, с появлением более зрелой версии пакета, для перехода на ее использование достаточно будет просто изменить инструкцию импортирования на `import bio "bio_v2"`, не трогая остального программного кода. Такой прием применим, только если обе версии, `bio_v1` и `bio_v2`, предоставляют одинаковый API (или если API версии `bio_v2` является надмножеством API версии `bio_v1`). С другой стороны, пакеты из стандартной библиотеки желательно не импортировать под псевдонимами, так как это может вызвать путаницу и раздражение у тех, кто будет сопровождать программу.

Как уже упоминалось в главе 5 (§5.6.2), в момент импортирования пакета вызывается его функция `init()` (если она имеется). В некоторых ситуациях не требуется явно использовать пакет, но желательно, чтобы его функция `init()` была выполнена.

Например, в программе обработки изображений может потребоваться зарегистрировать все графические форматы, поддерживаемые стандартной библиотекой Go, при этом ни одна функция из пакетов, предоставляющих поддержку этих форматов, в программе не используется. Ниже приводится инструкция `import` из файла `imgetag1.go` программы `imgetag1` (из упражнений в главе 7).

```
import (  
    "fmt"  
    "image"  
    "os"  
    "path/filepath"  
    "runtime"  
    _ "image/gif"  
    _ "image/jpeg"  
    _ "image/png"  
)
```

Здесь пакеты `image/gif`, `image/jpeg` и `image/png` импортируются исключительно с целью выполнить их функции `init()` (которые регистрируют поддерживаемые ими форматы в пакете `image`). Для каждого из этих пакетов в качестве псевдонима указан *пустой идентификатор*, поэтому компилятор Go не будет выдавать предупреждения о неиспользуемых пакетах.

9.2. Сторонние пакеты

Утилита `go`, использовавшаяся на протяжении всей книги для сборки программ и пакетов (например, пакета `map`), может также использоваться для загрузки, сборки и установки сторонних пакетов. (При этом, конечно, предполагается, что компьютер подключен к Интернету.) Список сторонних пакетов хранится на сайте godash-board.appspot.com/project. (Альтернативный способ заключается в том, чтобы загрузить исходные тексты, например непосредственно из распределенной системы управления версиями, и собрать пакет на локальном компьютере.)

Чтобы установить пакет, находящийся на сайте Go Dashboard, нужно щелкнуть на соответствующей ссылке, для перехода на домашнюю страницу пакета. Где-то на сайте пакета должно присутствовать описание команды `go get` для загрузки и установки пакета.

Например, если в списке на сайте Go Dashboard щелкнуть на ссылке **freetype-go**, откроется страница code.google.com/p/freetype-go/, где в центральной ее части демонстрируется (на момент написания этих строк) команда установки пакета: `go get code.google.com/p/freetype-go/freetype`.

Получив сторонний пакет, команда `go get` должна его куда-то установить. По умолчанию она устанавливает пакеты в первый каталог, указанный в переменной окружения `GOPATH`, если она определена, в противном случае пакет будет установлен в каталог `GOROOT`. Если пакет необходимо принудительно установить в каталог `GOROOT`, достаточно просто удалить переменную окружения `GOPATH` перед выполнением команды `go get`.

После запуска команда `go get` загрузит пакет, соберет его и установит. Ознакомиться с документацией вновь установленного пакета можно, запустив утилиту `godoc` в режиме веб-сервера (например, `godoc -http=:8000`) и открыв страницу с документацией пакета.

Чтобы избежать конфликтов, в качестве имен сторонних пакетов обычно используются доменные имена, гарантирующие их

уникальность. Например, чтобы задействовать пакет FreeType, его следует импортировать инструкцией `import "code.google.com/p/freetype-go/freetype"`. Конечно, когда дело дойдет до использования функций из пакета, достаточно будет использовать только последний компонент имени, например `font, err := freetype.ParseFont(fontdata)`. А в случае маловероятного конфликта при использовании последнего компонента имени всегда можно использовать псевдоним, например `import ftype "code.google.com/p/freetype-go/freetype"` и затем применять его в программном коде `font, err := ftype.ParseFont(fontdata)`.

Сторонние пакеты обычно поддерживают версию Go 1, но иногда они могут требовать более поздней версии, или для загрузки может быть доступно несколько версий, в том числе версии, требующие самых свежих версий Go, еще находящихся в разработке. Вообще, всегда лучше отдавать предпочтение стабильным версиям Go (например, Go 1) и использовать пакеты, совместимые с этими версиями.

9.3. Краткий обзор команд компилятора Go

В состав стандартного компилятора `gc` для языка Go входит не только комплект компиляторов и компоновщиков (`6g`, `6l` и др.), но также множество других инструментов. Наиболее часто используемой из них является утилита `go`, которая может использоваться для сборки программ и пакетов, для загрузки и установки сторонних программ и пакетов, а также для проведения модульного тестирования и измерения производительности, как было показано выше (§9.1.1.3). Команда `go help` выводит полный список доступных команд, а команда `go help имя_команды` — справку по указанной команде `имя_команды`. Имеется также инструмент `godoc`, предназначенный для отображения документации (см. врезку «Документация по языку Go» в главе 1).

В дополнение к инструментам, использовавшимся в этой книге, существует ряд других инструментов и команд утилиты `go`, которые будут упомянуты здесь. Одна из них — команда `go vet`, которая выполняет простейшие проверки на наличие ошибок в программах на языке Go, в частности в вызовах функций вывода из пакета `fmt`.

Еще одна команда — `go fix`. Иногда в новые версии Go вносятся изменения, касающиеся синтаксиса языка или, что случается чаще, прикладных интерфейсов библиотеки, которые вызывают нарушения в работе существующего программного кода. Выполнив команду `go`

`fix` для существующего программного кода, можно автоматически исправить имеющиеся несоответствия. Перед запуском команды `go fix` *настоятельно* рекомендуется сохранить все исходные тексты в системе управления версиями или, по крайней мере, создать резервную копию. Это позволит увидеть внесенные изменения и откатить любое из них или все сразу, если автоматическое исправление программного кода нарушит его работу. Можно также предварительно запустить команду `go fix -diff`, чтобы увидеть, какие изменения будут внесены без фактического их применения.

Последняя команда, которой мы коснемся, является команда `gofmt`. Она форматирует программный код на языке Go стандартизованным способом и рекомендуется к применению разработчиками Go. Достоинством команды `gofmt` является ее способность наилучшим образом отформатировать программный код и гарантировать единообразие его оформления. Весь программный код в книге был отформатирован с помощью утилиты `gofmt` (кроме строк длиннее 75 символов, в которые вручную были добавлены переносы на следующую строку, чтобы уместить их по ширине книжной страницы).

9.4. Краткий обзор стандартной библиотеки языка Go

Стандартная библиотека Go включает огромное число пакетов, обеспечивающих широчайшие функциональные возможности. Обзор, представленный ниже, очень краткий и выборочный. Весьма вероятно, что содержимое библиотеки будет продолжать увеличиваться и после публикации этой книги, поэтому для знакомства с API библиотеки лучше пользоваться электронной документацией (golang.org/pkg/) или использовать инструмент `godoc`, где можно найти самую свежую информацию и получить исчерпывающий перечень функциональных возможностей, доступных в каждом пакете.

В стандартной библиотеке существует «экспериментальный» пакет `exp`, где начинают свою жизнь пакеты, являющиеся *потенциальными* кандидатами на включение в стандартную библиотеку. Эти пакеты не следует использовать, только если вы не собираетесь принимать участия в их разработке (например, проводить тестирование, документирование или посылать свои исправления). Обычно пакет `exp` доступен, когда Go собирается из исходных текстов, извлекаемых из репозитория Google Go, но он может отсутствовать в уже

собранных пакетах. Все остальные пакеты могут использоваться без ограничений, однако на момент написания этих строк некоторые из них страдали неполнотой реализации.

9.4.1. Пакеты для работы с архивами и сжатыми файлами

Программы на языке Go могут читать и писать в тарболлы и zip-файлы. Для этих целей используются пакеты `archive/tar` и `archive/zip`, а для работы со сжатыми тарболлами дополнительно используются пакеты `compress/gzip` и `compress/bzip2`. Приемы использования всех этих пакетов демонстрировались в этой книге в примерах `pack` и `unpack` (§8.2).

В стандартной библиотеке имеется также поддержка других форматов сжатия. Например, формат Lempel-Ziv-Welch (`compress/lzw`), используемый для сжатия изображений `.tiff` и файлов документов `.pdf`.

9.4.2. Пакеты для работы с байтами и строками

Пакеты `bytes` и `strings` включают множество общих функций, только функции из первого пакета оперируют значениями типа `[]byte`, а функции из второго пакета – строковыми значениями. Пакет `strings` содержит утилиты, наиболее востребованные при работе со строками, используемые для поиска и замены подстрок, разбиения и усечения строк, и изменения регистра символов (§3.6.1). Пакет `strconv` содержит функции преобразования числовых и логических значений в строки и обратно (§3.6.2).

Пакет `fmt` предоставляет различные, чрезвычайно удобные функции вывода и сканирования. Функции вывода рассматривались в главе 3 (§3.5), а примеры их использования можно было увидеть на протяжении всей книги. Функции сканирования были перечислены в табл. 8.2, а некоторые примеры их использования – в §8.1.3.2.

Пакет `unicode` содержит функции для определения свойств символов, например является символ печатаемым или цифрой (§3.6.4). Пакеты `unicode/utf8` и `unicode/utf16` содержат функции декодирования и кодирования значений типа `rune` (то есть кодовых пунктов/символов Юникода). Описание пакета `unicode/utf8` можно найти в §3.6.3. Пример использования пакета `unicode/utf16` приводится в главе 8, в решении упражнения `utf16-to-utf8`.

Пакеты `text/template` и `html/template` можно использовать для создания шаблонов с целью сгенерировать вывод текстовой информации (например, в формате HTML) на основе подставляемых данных. Ниже приводится очень короткий пример использования пакета `text/template`.

```
type GiniIndex struct {
    Country string
    Index    float64
}
gini := []GiniIndex{{"Japan", 54.7}, {"China", 55.0}, {"U.S.A.", 80.1}}
giniTable := template.New("giniTable")
giniTable.Parse(
    `<TABLE>` +
        `{{range .}}` +
        `{{printf "<TR><TD>%s</TD><TD>%.1f%%</TD></TR>" .Country .Index}}` +
        `{{end}}` +
        `</TABLE>`)
err := giniTable.Execute(os.Stdout, gini)
<TABLE>
<TR><TD>Japan</TD><TD>54.7%</TD></TR>
<TR><TD>China</TD><TD>55.0%</TD></TR>
<TR><TD>U. S. A.</TD><TD>80.1%</TD></TR>
</TABLE>
```

Функция `template.New()` создает новое значение типа `*template`. `Template` с указанным именем. Имена шаблонов удобно использовать для идентификации шаблонов, вложенных в другие шаблоны. Функция `template.Template.Parse()` выполняет парсинг шаблона (обычно из HTML-файла), готового к использованию. Функция `template.Template.Execute()` применяет шаблон, выводя результаты в указанное значение типа `io.Writer` и читая исходные данные, предназначенные для наполнения шаблона, из своего второго аргумента. В данном примере вывод производится в поток `os.Stdout`, а роль исходных данных играет срез `gini` со структурами `GiniIndex`. (Для большей наглядности вывод был разбит на несколько строк.)

Выполняемые инструкции внутри шаблона заключены в двойные фигурные скобки (`{{` и `}}`). Инструкция `{{range}} ... {{end}}` может использоваться для выполнения итераций по всем элементам в срезе. Здесь оператор точки (`.`) соответствует текущему элементу типа `GiniIndex` в срезе. Доступ к экспортируемым полям структуры можно получить с помощью их имен с предшествующей точкой,

обозначающей текущий элемент. Инструкция `{{printf}}` действует подобно функции `fmt.Printf()`, только вместо круглых скобок и запятых, разделяющих аргументы, используются пробелы.

Пакеты `text/template` и `html/template` поддерживают собственный, весьма развитый язык шаблонов со множеством инструкций, включая циклы и условные инструкции, дающий возможность создавать переменные и вызывать методы и обладающий множеством других возможностей. Кроме того, пакет `html/template` устойчив против атак типа инъекция программного кода.

9.4.3. Пакеты для работы с коллекциями

Срезы являются самым эффективным типом коллекций в языке Go, но иногда бывает желательно или необходимо использовать коллекции более специализированных типов. Во многих случаях с успехом можно использовать встроенный тип `map`, однако в стандартной библиотеке имеется пакет `container`, включающий различные пакеты с реализациями различных типов коллекций.

Пакет `container/heap` предоставляет функции управления пирамидами (`heap`)¹, где роль пирамид могут играть значения, реализующие интерфейс `heap.Interface`, объявленный в пакете `heap`. Пирамида (строго говоря, *прямая пирамида* (`min-heap`)) поддерживает хранение своих элементов в отсортированном виде, когда первый элемент всегда имеет наименьшее значение (или наибольшее, в случае обратной пирамиды (`max-heap`)) – известное основное свойство пирамид. Интерфейс `heap.Interface` встраивает интерфейс `sort.Interface` и добавляет методы `Push()` и `Pop()`. (Интерфейс `sort.Interface` обсуждался в §4.2.4 и §5.7.)

Создать собственный тип пирамид, реализующий интерфейс `heap.Interface`, не составляет никакого труда. Ниже приводится пример использования такой пирамиды.

```
ints := &IntHeap{5, 1, 6, 7, 9, 8, 2, 4}
heap.Init(ints) // Упорядочить
ints.Push(9)    // IntHeap.Push() не соблюдает основного свойства пирамид
ints.Push(7)
ints.Push(3)
heap.Init(ints) // После изменения необходимо вновь выполнить упорядочение
for ints.Len() > 0 {
```

¹ Для обозначения пирамид иногда также используются термины «двоичная куча» и «сортирующее дерево». – *Прим. перев.*

```

    fmt.Printf("%v ", heap.Pop(ints))
}
fmt.Println() // выведет: 1 2 3 4 5 6 7 7 8 9 9

```

Ниже приводится полная реализация типа `IntHeap`.

```

type IntHeap []int
func (ints *IntHeap) Less(i, j int) bool {
    return (*ints)[i] < (*ints)[j]
}
func (ints *IntHeap) Swap(i, j int) {
    (*ints)[i], (*ints)[j] = (*ints)[j], (*ints)[i]
}
func (ints *IntHeap) Len() int {
    return len(*ints)
}
func (ints *IntHeap) Pop() interface{} {
    x := (*ints)[ints.Len()-1]
    *ints = (*ints)[:ints.Len()-1]
    return x
}
func (ints *IntHeap) Push(x interface{}) {
    *ints = append(*ints, x.(int))
}

```

Этой реализации вполне достаточно во многих ситуациях. Программный код можно сделать немного более удобочитаемым, если объявить тип как `type IntHeap struct { ints []int }`, потому что благодаря такому объявлению внутри методов можно ссылаться на срез как `ints.ints` вместо `*ints`.

Пакет `container/list` реализует двусвязные списки. Элементы добавляются в такой список как значения типа `interface{}`. При извлечении из списка элементы возвращаются как значения типа `list.Element`, где оригинальные значения доступны в виде поля `list.Element.Value`.

```

items := list.New()
for _, x := range strings.Split("ABCDEFGH", "") {
    items.PushFront(x)
}
items.PushBack(9)
for element := items.Front(); element != nil; element = element.Next() {
    switch value := element.Value.(type) {

```

```
    case string:
        fmt.Printf("%s ", value)
    case int:
        fmt.Printf("%d ", value)
}
fmt.Println() // выведет: H G F E D B A 9
```

В этом примере в начало нового списка было помещено восемь односимвольных строк, а потом в конец списка было добавлено целое число. Затем были выполнены итерации по элементам списка и выведены их значения. В действительности здесь можно не использовать инструкцию `switch`, а вывести значения вызовом `fmt.Printf("%v ", element.Value)`, но если бы в цикле потребовалось выполнять другие операции с элементами, помимо их вывода, тогда инструкция `switch` выбора по типу была бы совершенно необходима, если, конечно, список содержал бы элементы разных типов. Если все элементы списка являются значениями одного типа, тогда можно использовать операцию приведения типа, например `element.Value.(string)` для строковых элементов. (Операция выбора по типу рассматривалась в §5.2.2.2, а операция приведения типа – в §5.1.2.)

Помимо методов, показанных в примере выше, тип `list.List` предоставляет множество других методов, включая `Back()`, `Init()` (для очистки списка), `InsertAfter()`, `InsertBefore()`, `Len()`, `MoveToBack()`, `MoveToFront()`, `PushBackList()` (для добавления одного списка в конец другого) и `Remove()`.

В стандартной библиотеке имеется также пакет `container/ring`, реализующий кольцевой список¹.

Кроме различных типов коллекций, хранящих свои данные в памяти, в стандартной библиотеке Go имеется также пакет `database/sql`, предоставляющий обобщенный интерфейс к базам данных SQL. Для работы с фактическими базами данных необходимо отдельно устанавливать специализированные пакеты-драйверы. Эти пакеты наряду со многими другими пакетами коллекций доступны на сайте Go Dashboard (godashboard.appspot.com/project). И, как было показано выше, в загружаемых примерах к книге присутствует пакет, реализующий тип `omap.Map` упорядоченных отображений, основанных на левосторонних красно-черных деревьях (§6.5.3).

¹ В старых версиях Go может присутствовать пакет `container/vector`. Этот пакет считается устаревшим, и вместо него рекомендуется использовать срезы и встроенную функцию `append()` (§4.2).

9.4.4. Пакеты для работы с файлами и ресурсами операционной системы

Стандартная библиотека предоставляет множество пакетов для работы с файлами и каталогами и для взаимодействия с операционной системой. Во многих случаях эти пакеты реализуют абстракции, независимые от типа операционной системы, упрощающие разработку кросс-платформенных приложений на языке Go.

Пакет `os` («operating system» – операционная система) предоставляет функции для выполнения таких операций, как изменение текущего рабочего каталога, смена владельца файла и изменение разрешений на доступ к нему, получение и изменение значений переменных окружения, а также создание и удаление файлов и каталогов. Кроме того, в данном пакете имеются функции для создания и открытия файлов (`os.Create()` и `os.Open()`) и для получения атрибутов (например, в виде значения типа `os.FileInfo`), каждая из которых уже встречалась в предыдущих главах. (Например, см. §7.2.5 и главу 8.)

После открытия файла, что особенно характерно при работе с текстовыми файлами, часто бывает желательно обеспечить доступ к нему через буфер (например, чтобы читать его содержимое в виде строк, а не срезов с байтами). Необходимая в таких случаях функциональность предоставляется пакетом `bufio`, и снова в предыдущих главах было представлено множество примеров использования этого пакета. В дополнение к типам `bufio.Reader` и `bufio.Writer`, предназначенным для чтения и записи строк, предоставляется также возможность читать значения типа `rune`, одиночные байты и последовательности байт, а также записывать их.

Пакет `io` («input/output» – ввод/вывод) предоставляет огромное количество функций для работы с интерфейсами `io.Reader` и `io.Writer`. (Оба эти интерфейса поддерживаются значениями типа `*os.File`.) Например, функция `io.Copy()` использовалась для копирования данных (§8.2.1). Кроме того, этот пакет содержит функции для создания синхронных каналов в памяти.

Пакет `io/ioutil` предоставляет несколько высокоуровневых вспомогательных функций. В том числе функцию `ioutil.ReadAll()`, которая читает все данные из значения, поддерживающего интерфейс `io.Reader`, и возвращает их в виде среза `[]byte`; `ioutil.ReadFile()`, которая делает то же самое, но принимает строковый аргумент (имя файла), а не значение с интерфейсом `io.Reader`; `ioutil.TempFile()`,

которая возвращает временный файл (значение типа `*os.File`); и `ioutil.WriteFile()`, которая записывает срез типа `[]byte` в файл с указанным именем.

Пакет `path` содержит функции для работы с путями в стиле Unix, адресами URL, «ссылками» `git`¹, файлами на FTP-серверах и т. д. Пакет `path/filepath` предоставляет те же самые функции, что и пакет `path` (и множество других функций), предназначенные для работы с путями платформенно независимым способом. Кроме того, в этом пакете имеется функция `filepath.Walk()`, выполняющая рекурсивный обход всех файлов и каталогов в указанном пути, как было показано в главе 7 (§7.2.5).

Пакет `runtime` содержит массу функций и типов, обеспечивающих доступ к среде времени выполнения языка Go. Большинство из них слишком узкоспециализированы, и их применение не требуется при создании типичных программ на языке Go. Однако пара констант из этого пакета может пригодиться, например `runtime.GOOS`, хранящая строку (например, `"darwin"`, `"freebsd"`, `"linux"` или `"windows"`), и `runtime.GOARCH`, так же хранящая строку (например, `"386"`, `"amd64"` или `"arm"`). Функция `runtime.GOROOT()` возвращает значение переменной окружения `GOROOT` (или корневой каталог сборки Go, если эта переменная окружения не определена), а функция `runtime.Version()` возвращает версию Go (в виде строки). В главе 7 было также показано, как можно воспользоваться функциями `runtime.GOMAXPROCS()` и `runtime.NumCPU()`, чтобы обеспечить использование приложением всех процессоров компьютера.

9.4.4.1. Пакеты, имеющие отношение к форматам файлов

Язык Go обладает великолепными инструментами для работы с текстовыми (в 7-битной кодировке ASCII или в кодировках UTF-8 и UTF-16) и двоичными файлами. В стандартной библиотеке Go имеются специализированные пакеты для работы с файлами в формате JSON и XML, а также с файлами в собственном, очень быстром, компактном и удобном двоичном формате Go. (Все эти форматы плюс пользовательские двоичные форматы рассматривались в главе 8.)

Дополнительно в стандартной библиотеке имеется пакет `encoding/csv` для чтения CSV-файлов («comma-separated values» – значения,

¹ Здесь имеются в виду ссылки в распределенной системе управления версиями GIT. – Прим. перев.

разделенные запятыми). Этот пакет интерпретирует строки в таких файлах, как записи, состоящие из полей (разделенных запятыми). Пакет достаточно универсален. Например, он позволяет изменять символ-разделитель (использовать, например, точку, символ табуляции или любой другой символ), а также другие параметры, определяющие порядок чтения и записи данных.

Пакет `encoding` содержит несколько пакетов, один из которых, `encoding/binary`, уже использовался в книге для чтения и записи двоичных файлов (§8.1.5). Другие пакеты реализуют механизмы преобразования данных в другие форматы и обратно, например пакет `encoding/base64` можно использовать для кодирования/декодирования адресов URL, часто использующих этот формат.

9.4.5. Пакеты для работы с графикой

Пакет `image` предоставляет некоторые высокоуровневые функции для создания и хранения изображений. Кроме того, в его состав входят несколько пакетов, обеспечивающих возможность преобразований между различными стандартными графическими форматами файлов, такие как `image/jpeg` и `image/png`. Некоторые из них уже рассматривались выше в этой главе (§9.1.2) и в одном из упражнений в главе 7.

Пакет `image/draw` содержит несколько простых функций рисования, как было показано в главе 6 (§6.5.2). Еще больше функций рисования предоставляет сторонний пакет `freetype`. Сам пакет `freetype` может использоваться для рисования текста с использованием любого шрифта `TrueType`, а пакет `freetype/raster` – для рисования прямых, а также кубических и квадратичных кривых. (Порядок получения и установки пакета `freetype` описывался выше; §9.2.)

9.4.6. Математические пакеты

Пакет `math/big` предоставляет возможность использовать целые (`big.Int`) и рациональные (`big.Rat`) числа неограниченного размера (размер ограничивается только объемом доступной памяти). Эти типы чисел уже рассматривались ранее (§2.3). Порядок использования чисел типа `big.Int` демонстрируется в примере `pi_by_digits` (§2.3.1.1). Пакет `math/big` также содержит функцию `big.ProbablyPrime()`.

В пакете `math` имеются все стандартные математические функции (выполняющие операции с числами типа `float64`) и несколько стандартных констант (см. табл. 2.8, табл. 2.9 и табл. 2.10).

Пакет `math/cmplx` содержит несколько стандартных функций для работы с комплексными числами (значениями типа `complex128`). См. табл. 2.11.

9.4.7. Различные пакеты

В дополнение к пакетам, которые можно грубо отнести к какой-либо группе пакетов, в стандартной библиотеке имеется ряд пакетов, стоящих особняком.

Пакет `crypto` может использоваться для вычисления контрольных сумм с использованием алгоритмов MD5, SHA-1, SHA-224, SHA-256, SHA-384 и SHA-512. (Поддержка каждого алгоритма реализована в виде отдельного пакета, например `crypto/sha512`.) Кроме того, пакет `crypto` содержит пакеты, обеспечивающие возможность шифрования и дешифрования с использованием различных алгоритмов, таких как AES, DES и др., реализованных в виде отдельных пакетов с соответствующими именами (например, `crypto/aes`, `crypto/des`).

Пакет `exec` используется для запуска внешних программ. То же самое можно делать с помощью функции `os.StartProcess()`, но тип `exec.Cmd` гораздо удобнее в использовании.

Пакет `flag` предоставляет парсер аргументов командной строки. Он принимает параметры в стиле X11 (например, `-width`, отличающиеся от параметров в стиле GNU `-w` и `--width`). Пакет способен выводить простейшие сообщения о порядке использования и не предоставляет никаких дополнительных проверок, кроме проверки типа значения. (То есть средствами пакета можно, например, проверить, что аргумент является целым числом, но нельзя ограничить диапазон допустимых значений.) Другие альтернативные реализации можно найти на сайте Go Dashboard (godashboard.appspot.com/project).

Пакет `log` предоставляет функции для регистрации информации в журнале (по умолчанию в качестве журнала используется `os.Stdout`), для завершения программ или возбуждения аварийной ситуации с выводом сообщения в журнал. Место, куда пакет `log` будет выводить информацию, можно изменить, передав функции `log.SetOutput()` любое значение, поддерживающее интерфейс `io.Writer`. При выводе сообщений сначала выводятся дата и время, а затем текст самого сообщения. Вывод даты и времени можно подавить вызовом функции `log.SetFlags(0)` перед первым вызовом функции вывода. Имеется также возможность с помощью функции `log.New()` создавать собственные значения для вывода в журнал.

Пакет `math/rand` предоставляет множество функций, позволяющих генерировать последовательности псевдослучайных чисел, включая `rand.Int()`, возвращающую случайное целое, и `rand.Intn(n)`, возвращающую случайное целое в диапазоне `[0, n)`. В пакете `crypto/rand` имеется функция, генерирующая криптостойкие псевдослучайные числа.

Пакет `regexp` предоставляет очень быстрый и мощный механизм регулярных выражений, поддерживающий синтаксис RE2. Этот пакет уже демонстрировался в нескольких примерах в книге, хотя там преднамеренно применялись простые регулярные выражения и не использовались все возможности, имеющиеся в пакете, чтобы не отвлекаться от основной темы. Пакет был представлен в главе 3 (§3.6.5).

Пакет `sort` предоставляет удобные функции для сортировки срезов с числами и строками, для выполнения быстрого поиска (методом дихотомии) в таких отсортированных срезах. В нем также имеются обобщенные функции `sort.Sort()` и `sort.Search()` для работы с данными пользовательских типов (см. §4.2.4, табл. 4.2, и §5.6.7.)

Пакет `time` содержит функции для измерения интервалов времени и парсинга строковых значений с датой и временем. Функция `time.After()` может использоваться для передачи текущего времени в возвращаемый ею канал спустя указанный промежуток времени в наносекундах, как было показано ранее (§7.2.2). Функции `time.Tick()` и `time.NewTicker()` могут использоваться для создания «тикающего» канала, в который через указанные промежутки времени будет выполняться посылка. Тип `time.Time` имеет методы для определения текущего времени, для форматирования даты и времени в виде строки, для парсинга строкового представления даты и времени. (Примеры использования значений типа `time.Time` были представлены в главе 8).

9.4.8. Пакеты для работы с сетью

Стандартная библиотека Go содержит множество пакетов для создания сетевых приложений. Пакет `net` предоставляет функции и типы для организации обмена данными через сетевые сокеты и доменные сокеты Unix с использованием протоколов TCP/IP и UDP. Пакет также предоставляет функции для разрешения доменных имен.

Пакет `net/http` используется пакетом `net` и обеспечивает возможность парсинга HTTP-запросов и ответов, а также реализует простейшего HTTP-клиента. Кроме того, пакет `net/http` включает

реализацию легко расширяемого HTTP-сервера, как было показано в главе 2 (§2.4) и в упражнениях в главе 3. Пакет `net/url` реализует парсинг адресов URL и экранирование запросов.

В стандартной библиотеке имеется также ряд других высокоуровневых пакетов для работы с сетью. Один из них – пакет `net/rpc` («Remote Procedure Call» – вызов удаленных процедур), позволяющий серверу создавать объекты с экспортируемыми методами, доступными для вызова со стороны клиента. Другой – пакет `net/smtp` («Simple Mail Transport Protocol» – упрощенный протокол электронной почты), который может использоваться для отправки электронной почты.

9.4.9. Пакет *reflect*

Пакет `reflect` реализует возможность рефлексии (также называется *интроспекцией*) во время выполнения, то есть возможность получать доступ и взаимодействовать со значениями любых типов.

Пакет также предоставляет ряд вспомогательных функций, таких как `reflect.DeepEqual()`, способная сравнивать любые два значения, например срезы, которые нельзя сравнить с помощью операторов `==` и `!=`.

Каждое значение в языке Go имеет два атрибута: его фактическое значение и его тип. Функция `reflect.TypeOf()` возвращает тип любого значения.

```
x := 8.6
y := float32(2.5)
fmt.Printf("var x %v = %v\n", reflect.TypeOf(x), x)
fmt.Printf("var y %v = %v\n", reflect.TypeOf(y), y)
var x float64 = 8.6
var y float32 = 2.5
```

Здесь с использованием механизма рефлексии выводятся значения двух вещественных переменных и их типы в виде объявлений `var`.

Функция `reflect.ValueOf()` возвращает значение типа `reflect.Value`, содержащее ссылку на указанное значение, а не его фактическое значение. Чтобы получить доступ к фактическому значению, необходимо использовать один из методов типа `reflect.Value`.

```
word := "Chameleon"
value := reflect.ValueOf(word)
```

```
text := value.String()
fmt.Println(text)
Chameleon
```

Тип `reflect.Value` имеет множество методов, позволяющих извлекать фактическое значение, включая `reflect.Value.Bool()`, `reflect.Value.Complex()`, `reflect.Value.Float()`, `reflect.Value.Int()` и `reflect.Value.String()`.

Пакет `reflect` можно также использовать для работы с типами коллекций, такими как срезы и отображения, а также со структурами — с его помощью можно даже получить доступ к тегам полей структуры. (Данная возможность используется в пакетах `json` и `xml`, как было показано в главе 8.)

```
type Contact struct {
    Name string "check:len(3,40)"
    Id    int    "check:range(1,999999)"
}
person := Contact{"Bjork", 0xDEDED}
personType := reflect.TypeOf(person)
if nameField, ok := personType.FieldByName("Name"); ok {
    fmt.Printf("%q %q %q\n", nameField.Type, nameField.Name, nameField.Tag)
}
"string" "Name" "check:len(3,40)"
```

Фактическое значение, хранящееся в значении типа `reflect.Value` можно изменить, *если* оно допускает такую возможность. Возможность изменения проверяется вызовом метода `reflect.Value.CanSet()`, возвращающим значение типа `bool`.

```
presidents := []string{"Obama", "Bushy", "Clinton"}
sliceValue := reflect.ValueOf(presidents)
value = sliceValue.Index(1)
value.SetString("Bush")
fmt.Println(presidents)
[Obama Bush Clinton]
```

Несмотря на то что строки в языке Go являются неизменяемыми значениями, любой элемент в срезе типа `[]string` можно заменить другой строкой, именно это и делается в примере выше. (Естественно, в данном конкретном примере проще всего было бы использовать инструкцию `presidents[1] = "Bush"` и вообще не использовать механизма интроспекции.)

Само неизменяемое значение изменить невозможно, но, имея адрес оригинального значения, его можно заменить другим значением.

```
count := 1
if value = reflect.ValueOf(count); value.CanSet() {
    value.SetInt(2) // Возбудит аварийную ситуацию! Нельзя изменить значение int.
}
fmt.Print(count, " ")
value = reflect.ValueOf(&count)
// Здесь нельзя вызвать метод SetInt() значения value,
// потому что оно имеет тип *int, а не int
pointee := value.Elem()
pointee.SetInt(3) // OK. Значение, на которое ссылается указатель, заменить можно.
fmt.Println(count)
1 3
```

Вывод этого фрагмента показывает, что если условное выражение в инструкции `if` вернет `false`, тело этой инструкции не будет выполнено. Несмотря на невозможность изменить значения неизменяемых типов, таких как `int`, `float64` или `string`, можно воспользоваться методом `reflect.Value.Elem()`, чтобы из значения типа `reflect.Value` извлечь указатель на фактическое значение и заменить его, что и делается в конце этого примера.

Механизм рефлексии можно также использовать для вызова произвольных функций и методов. Ниже приводится пример вызова пользовательской функции `TitleCase()` (здесь не показана). Первый вызов осуществляется как обычно, а второй – с применением механизма рефлексии.

```
caption := "greg egan's dark integers"
title := TitleCase(caption)
fmt.Println(title)
titleFuncValue := reflect.ValueOf(TitleCase)
values := titleFuncValue.Call([]reflect.Value{reflect.ValueOf(caption)})
title = values[0].String()
fmt.Println(title)
Greg Egan's Dark Integers
Greg Egan's Dark Integers
```

Метод `reflect.Value.Call()` принимает и возвращает срез типа `[]reflect.Value`. В данном случае ему передается единственное значение (то есть срез с длиной, равной 1), и в результате получается единственное значение.

Методы вызываются аналогично, а кроме того, имеется возможность сначала проверить наличие метода и только потом вызвать его.

```

a := list.New()                // a.Len() == 0
b := list.New()
b.PushFront(1)                 // b.Len() == 1
c := stack.Stack{}
c.Push(0.5)
c.Push(1.5)                    // c.Len() == 2
d := map[string]int{"A": 1, "B": 2, "C": 3} // len(d) == 3
e := "Four"                     // len(e) == 4
f := []int{5, 0, 4, 1, 3}       // len(f) == 5
fmt.Println(len(a), len(b), len(c), len(d), len(e), len(f))
0 1 2 3 4 5

```

Здесь создаются два списка (с использованием пакета `container/list`), в один из которых добавляется элемент. Затем создается стек (с использованием пакета `stack/stack`, созданного в главе 1; §1.5), и в него добавляются два элемента. Далее создаются отображение, строка и срез с целыми числами, все различной длины. Затем с помощью пользовательской обобщенной функции `Len()` определяются их длины.

```

func Len(x interface{}) int {
    value := reflect.ValueOf(x)
    switch reflect.TypeOf(x).Kind() {
    case reflect.Array, reflect.Chan, reflect.Map, reflect.Slice,
         reflect.String:
        return value.Len()
    default:
        if method := value.MethodByName("Len"); method.IsValid() {
            values := method.Call(nil)
            return int(values[0].Int())
        }
    }
    panic(fmt.Sprintf("'%'v' does not have a length", x))
}

```

Эта функция возвращает длину переданного ей значения или возбуждает аварийную ситуацию, если тип значения не поддерживает понятия длины.

Начинается функция с получения значения в виде значения типа `reflect.Value`, которое потребуется далее. Затем производится

выбор в зависимости от типа значения, возвращаемого методом `reflect.Type.Kind()`. Если значение принадлежит одному из типов, поддерживающих встроенную функцию `len()`, метод `reflect.Value.Len()` можно вызвать непосредственно. В противном случае либо значение не поддерживает понятия длины, либо его тип реализует метод `Len()`. Вызовом метода `reflect.Value.MethodByName()` функция получает метод или недопустимое значение `reflect.Value`. Если метод допустим, он вызывается. В данном случае методу не требуется передавать никаких аргументов, потому что в соответствии с соглашениями методы `Len()` не принимают аргументов.

Метод `reflect.Value.MethodByName()` возвращает значение типа `reflect.Value`, хранящее и метод, и значение. Поэтому при вызове `reflect.Value.Call()`, значение передается методу в качестве приемника.

Метод `reflect.Value.Int()` возвращает значение типа `int64`. Здесь оно преобразуется в значение типа `int`, чтобы привести к типу возвращаемого значения обобщенной функции `Len()`.

Если переданное значение не поддерживает встроенную функцию `len()` и не имеет метода `Len()`, обобщенная функция `Len()` возбудит аварийную ситуацию. Эту ошибку можно было бы обрабатывать иначе, например возвращая `-1`, чтобы указать, что «длина недоступна», или возвращая два значения, типов `int` и `error`.

Пакет `reflect` обладает весьма внушительными возможностями и позволяет выполнять операции в зависимости от динамического состояния программы. Однако, как отмечает Роб Пайк (Rob Pike), механизм рефлексии – это «мощный инструмент, требующий осторожного обращения, который не следует применять без явной необходимости»¹.

9.5. Упражнения

В этой главе предлагается выполнить три упражнения. Первое связано с созданием небольшого пакета. Второе – с созданием тестов для пакета. И в третьем упражнении нужно будет написать программу, использующую пакет. Сложность упражнений постепенно растет от первого к последнему, и последнее упражнение является достаточно сложным.

¹ Роб Пайк (Rob Pike) разместил в блоге интересную статью о механизме рефлексии в языке Go, которую можно найти по адресу: blog.golang.org/2011/09/laws-of-reflection.html.

1. Создайте пакет с именем, например, `my_linkutil` (в файле `my_linkutil/my_linkutil.go`). Пакет должен содержать две функции. Первая, `LinksFromURL(string) ([]string, error)`, должна принимать строку URL (например, `"http://www.qtrac.eu/index.html"`) и возвращать срез со всеми уникальными ссылками на странице (то есть значениями атрибута `href` тегов `<a>`) и значение `nil` (или `nil` и признак ошибки). Вторая функция, `LinksFromReader(io.Reader) ([]string, error)`, должна делать то же самое, но читать данные из значения типа `io.Reader` (например, из открытого файла или из значения типа `http.Response.Body`). В своей работе функция `LinksFromURL()` должна использовать функцию `LinksFromReader()`.

Одно из возможных решений приводится в файле `linkcheck/linkutil/linkutil.go`. Первая функция в решении занимает около 11 строк и использует функцию `http.Get()` из пакета `the net/http`. Вторая функция занимает примерно 16 строк и использует функцию `regexp.Regexp.FindAllSubmatch()`.

2. Стандартная библиотека Go предоставляет поддержку тестирования взаимодействий по протоколу HTTP (например, пакет `net/http/httptest`), но в данном упражнении достаточно будет реализовать проверку функции `my_linkutil.LinksFromReader()`, созданной в предыдущем упражнении. Для этой цели требуется создать тестовый файл (например, `my_linkutil/my_linkutil_test.go`), содержащий единственный тест, `TestLinksFromReader(*testing.T)`. Тест должен читать содержимое HTML-файла, хранящегося в файловой системе, содержимое другого файла, содержащего все уникальные ссылки из первого файла, и сравнивать ссылки, найденные в HTML-файле функцией `my_linkutil.LinksFromReader()` со ссылками во втором файле. Для нужд тестирования можно воспользоваться файлами `linkcheck/linkutil/index.html` и `linkcheck/linkutil/index.links`, находящимися в каталоге `my_linkutil`.

Одно из возможных решений приводится в файле `linkcheck/linkutil/linkutil_test.go`. Тестовая функция в этом решении занимает около 40 строк и использует функцию `sort.Strings()` для упорядочения найденных и ожидаемых ссылок, а также функцию `reflect.DeepEqual()` для их сравнения. Чтобы помочь тем, кто будет проводить тестирование, тестовая функция выводит первые несовпавшие ссылки.

3. Напишите программу, например, с именем `my_linkcheck`, которая будет принимать единственный адрес URL из командной строки (с префиксом `http://` или без него) и проверять допустимость каждой ссылки в указанной странице. Программа должна рекурсивно проверять каждую страницу, на которую ссылается данная страница, но, исключая ссылки, не являющиеся HTTP-ссылками, исключая файлы, не являющиеся HTML-файлами, и исключая ссылки на другие сайты. Каждая страница должна проверяться в отдельной go-подпрограмме – это приведет к созданию множества параллельных сетевых подключений, что позволит увеличить скорость выполнения, по сравнению с реализацией, выполняющей доступ к страницам последовательно. Естественно, на разных страницах могут присутствовать одинаковые ссылки, и такие ссылки должны проверяться только один раз. Конечно, программа должна использовать пакет `my_linkutil`, разработанный в первом упражнении.

Одно из возможных решений приводится в файле `linkcheck/linkcheck.go` и занимает около 150 строк. Чтобы избежать повторной проверки ссылок, в решении используется отображение с уже проверенными адресами URL. Отображение хранится внутри отдельной go-подпрограммы, для взаимодействия с которой используются три канала: один канал используется для добавления адресов URL, один – для отправки запросов на проверку присутствия URL в отображении и один – для ответов на запросы. (При желании можно воспользоваться поточно-ориентированным отображением из главы 7.) Ниже показан фрагмент вывода, произведенного командой `linkcheck www.qtrac.eu` (где множество строк было удалено полностью или частично).

```
+ read http://www.qtrac.eu
...
+ read http://www.qtrac.eu/gobook.html
+ read http://www.qtrac.eu/gobook-errata.html
...
+ read http://www.qtrac.eu/comparepdf.html
+ read http://www.qtrac.eu/index.html
...
+ links on http://www.qtrac.eu/index.html
  + checked http://ptgmedia.pearsoncmg.com/.../python/python2python3.pdf
  + checked http://www.froglogic.com
```




- can't check non-http link: <mailto:someone@somewhere.com>
- + checked <http://savannah.nongnu.org/projects/lout/>
- + read <http://www.qtrac.eu/py3book-errata.html>
- + links on <http://www.qtrac.eu>
 - + checked <http://endsoftpatents.org/innovating-without-patents>
- + links on <http://www.qtrac.eu/gobook.html>
 - + checked <http://golang.org>
 - + checked <http://www.qtrac.eu/gobook.html#eg>
 - + checked <http://www.informit.com/store/product.aspx?isbn=0321680561>
 - + checked <http://safari.informit.com/9780321680563>
 - + checked <http://www.qtrac.eu/gobook.tar.gz>
 - + checked <http://www.qtrac.eu/gobook.zip>
- can't check non-http link: <ftp://ftp.cs.usyd.edu.au/jeff/lout/>
- + checked <http://safari.informit.com/9780132764100>
- + checked <http://www.qtrac.eu/gobook.html#toc>
- + checked <http://www.informit.com/store/product.aspx?isbn=0321774637>

...



А. Эпилог

Разработчики языка Go долго и пристально изучали некоторые из наиболее широко используемых языков программирования, чтобы определить, какие особенности действительно будут полезны и продуктивны, а какие являются избыточными или даже контрпродуктивными. Они также учли свой коллективный опыт программирования, накапливавшийся в течение многих десятилетий. В результате на свет появился язык программирования Go.

С точки зрения языков Objective-C и C++, язык Go является объектно-ориентированным, «улучшенным» языком C. Как и язык Java, Go имеет свой собственный синтаксис, поэтому ему не требуется обеспечивать совместимость с C, как это вынуждены делать языки Objective-C и C++. Но, в отличие от Java, Go является компилируемым языком и потому не ограничен скоростью работы виртуальной машины.

Помимо иного подхода к объектно-ориентированному программированию, с его упором на абстрактные интерфейсы и конкретные типы, поддерживающие встраивание и агрегирование, язык Go также поддерживает дополнительные особенности, такие как литералы функций и замыкания. А встроенные типы отображений и срезов способны удовлетворить практически любые требования, предъявляемые к структурам данных. Строковый тип, основанный на Юникоде, использует кодировку UTF-8, де-факто ставшую международным стандартом, а стандартная библиотека предоставляет великолепную поддержку для работы с отдельными байтами и символами.

Язык Go обладает выдающейся поддержкой параллельного программирования. Его легковесные go-подпрограммы и высокоуровневые каналы существенно упрощают разработку многопоточных программ в сравнении со многими другими языками (такими как C, C++ или Java). А молниеносная компиляция программ на языке Go – как глоток свежего воздуха, особенно для тех, кому приходилось заниматься сборкой крупных программ и библиотек на языке C++.

Язык Go уже используется в различных коммерческих и некоммерческих организациях. И он используется внутри самой компании

Google, наряду с языками Java и Python, как язык разработки веб-приложений на основе Google App Engine (code.google.com/appengine/docs/go/overview.html).

Язык продолжает развиваться очень быстро, и во многом благодаря инструменту `go fix` программный код легко привести в соответствие с последней версией Go. Кроме того, разработчики Go предполагают сохранить обратную совместимость всех версий Go 1.x с версией Go 1, чтобы гарантировать стабильность и высокое качество языка.

Стандартная библиотека Go обеспечивает широчайшие функциональные возможности, но, даже когда обнаруживается, что в ней отсутствует необходимая функциональность, всегда можно попробовать отыскать недостающие пакеты на сайте Go Dashboard (go-dashboard.appspot.com/project) или, в некоторых случаях, можно использовать внешние библиотеки, написанные на других языках. Основным источником самой свежей информации о языке Go является сайт golang.org. Здесь всегда можно найти документацию для текущей версии, спецификации языка (написанные весьма живым языком), блоги, видеоролики и множество других документов.

Большинство из тех, кто приступает к изучению Go, уже имеют опыт программирования на других языках (таких как C++, Java, Python) и отягощены опытом объектно-ориентированного программирования на основе наследования. В языке Go преднамеренно не была реализована поддержка наследования. В то время как программы, например, на языках C++ и Java относительно просто можно перенести с одного языка на другой, в случае с языком Go лучше всего вернуться к основам, для чего был написан программный код, и переписать программу «с нуля». Пожалуй, самое важное отличие заключается в том, что языки, основанные на наследовании, позволяют смешивать код и данные, тогда как Go вынуждает разделять их. Такое разделение позволяет добиться высочайшей гибкости и упрощает создание многопоточных программ, но, чтобы адаптироваться к особенностям программирования на языке Go, программистам с опытом использования языков, основанных на наследовании, необходимы время и практика. Расс Кокс (Russ Cox), один из основных разработчиков Go, сказал:

«Очень жаль, что каждый раз, когда кто-то спрашивает о наследовании, ему отвечают: “ну есть же встраивание”. Встраивание – полезный прием и может рассматриваться как разновидность наследования, но когда кому-то требуется именно наследование, то

*встраивание не может считаться заменой. Правильный ответ: “вы думаете на C++, Python, Java, Eiffel или каком-то другом языке. Пере-
станьте! Думайте на Go!”»*

Go – очаровательный язык, простой в изучении и использовании, программировать на котором – одно удовольствие. Программисты на Go наверняка сочтут полезным для себя присоединиться к списку рассылки Go – в нем принимают участие много опытных разработчиков, и это идеальное место для дискуссий и решения проблем (groups.google.com/group/golang-nuts). А поскольку разработка языка Go ведется открыто, можно стать одним из разработчиков Go и оказывать помощь в сопровождении, улучшении и расширении языка (golang.org/doc/contribute.html).



В. Опасность патентов на программное обеспечение

Патенты являются весьма странной аномалией в капиталистических экономиках, потому что они обеспечивают частную монополию при посредничестве государства. Адам Смит (Adam Smith) резко осудил монополии в своей книге «The Wealth of Nations».

В наше время патенты пользуются широкой поддержкой различных предприятий – от маленьких производителей пылесосов до гигантских фармацевтических компаний. Но когда дело доходит до патентов на программное обеспечение, трудно найти человека, который положительно относился бы к патентам, кроме разве что патентных троллей (компаний, которые покупают и продают права на патенты, сами ничего не производя) и их адвокатов. Еще в 1991 году Билл Гейтс (Bill Gates) сказал: «Если бы в те времена, когда было придумано большинство сегодняшних идей, люди понимали, что могут дать патенты, и получали бы их, современная промышленность полностью остановилась бы». В современной обстановке его слова наполняются новым смыслом.

Программные патенты затрагивают любое предприятие, производящее программное обеспечение, на продажу или для внутреннего использования. Даже такие гиганты, как Kraft Foods и Ford Motor Co., не имеющие никакого отношения к разработке программного обеспечения, вынуждены тратить огромные суммы на патентные тяжбы в судах. Рискует даже каждый отдельный программист. Например, связанные списки, оказывается, запатентованы, но не их изобретателями, Алленом Ньюэллом (Allen Newell), Клиффом Шоу (Cliff Shaw) и Гербертом Симоном (Herbert Simon), когда они пришли к этой идее еще в 1955–1956 годах, а кем-то другим, 50 лет спустя (www.google.com/patents/about?id=26AJAAAAEBAJ&dq=linked+list). То же произошло со списками с пропусками, изобретенными Вильямом Пью (William Pugh) в 1990 году и запатентованными кем-то другим, десятилетие спустя. К сожалению, существуют десятки

тысяч других патентов на программное обеспечение, которые можно было бы привести в качестве примеров. Впрочем, упомяну еще один: «Система и метод, вынуждающие компьютер идентифицировать структуры данных и выполнять операции над ними», полученный компанией Apple в 1999 году и под действие которого подпадает любое программное обеспечение, манипулирующее структурами данных (www.google.com/patents?id=aFEWAAAAEBAJ&dq=5,946,647).

Такие слишком широкие и очевидные патенты легко можно было бы признать недействительными, но на практике даже такие гиганты, как Google, вынуждены платить миллионы долларов на оплату услуг юристов для защиты самих себя. Разве в такой ситуации могут начинающие, небольшие и средние компании, занимающиеся производством программного обеспечения, выпускать на рынок инновационные программы, не рискуя попасть под давление патентных троллей, живущих за чужой счет?

Вот как действует законодательная система в США и в других странах, имеющих аналогичные патентные законодательства. Прежде всего следует помнить, что патенты подлежат соблюдению, даже если «правонарушитель» не знает о существовании патента. Кроме того, штрафы за нарушение патента могут быть просто астрономическими. Теперь представьте разработчиков, работающих над закрытым программным кодом, которые самостоятельно разработали алгоритм выполнения некоторой операции. Патентный тролль, услышав, что компания-разработчик придумала какие-то инновации, которые могут принести прибыль, подает в суд на разработчиков за нарушение одного из своих универсальных патентов (такого как патент компании Apple, упомянутый выше). После этого исходные тексты программ необходимо будет представить для независимой экспертизы. Естественно, экспертиза будет проводиться не только на предмет нарушения указанного патента, но и всех патентов, которыми владеет тролль. Таким способом тролль получает возможность увидеть весь алгоритм, придуманный разработчиками, и может даже попытаться запатентовать его — в конце концов, в отличие от небольших компаний, он может позволить себе платить хорошие деньги своим юристам. Конечно, тролли не очень любят ходить в суд, потому что их «бизнес-модель» основана на вымогательстве: они предпочитают, чтобы производитель выпустил на рынок свою продукцию и выплачивал патентные отчисления за универсальный патент, который, по словам тролля, был нарушен. Большинство патентов, если не все, не имеют законной силы, в том смысле что

они этого не стоят, но тяжбы в судах привели бы к банкротству большинства малых компаний, поэтому они предпочитают платить патентные отчисления. Но, как только компании начинают платить патентные отчисления, это дает троллю дополнительную законную силу, он потом может процитировать своей следующей жертве список компаний, выплачивающих патентные отчисления.

Крупные компании могут позволить себе приобретать патенты и защищать себя от патентных троллей, и необязательно должны испытывать симпатии к малым предприятиям, которые рано или поздно могут вырасти и превратиться в конкурентов, поэтому большинство из них мало заботит сложившаяся ситуация. Некоторые компании, включая компании, занимающиеся разработкой открытого программного обеспечения, такие как Red Hat, приобретают в качестве защиты патенты на программное обеспечение, которые можно использовать для перекрестного лицензирования и минимизации юридических издержек. Насколько эффективными будут подобные действия, когда такие гиганты, как Apple, Google и Microsoft, владеют патентными портфелями на миллиарды долларов, пока остается только гадать. Компании, сделавшие такие огромные инвестиции, вряд ли будут способствовать приближению конца патентного права, пусть и разрушительного, так как это повлечет списание огромных средств, которое трудно будет оправдать перед акционерами или руководителями, чьи гонорары зависят от цен на акции компании.

Малые компании и частные предприниматели обычно не имеют серьезных финансовых мускулов для защиты от посягательств патентных троллей. Очень немногие могут попытаться счастья, переехав за рубеж, тогда как большинство в конечном итоге вынуждены будут заплатить серьезные деньги за защиту (или уйти из бизнеса) или выплачивать лицензионные отчисления за ничтожные патенты. Патенты на программное обеспечение уже оказывают негативное влияние на индивидуальных разработчиков и малые компании в США, усложняя и удорожая их бизнес, тем самым снижая их способность к расширению и созданию новых рабочих мест для программистов. Прибыль от патентных тяжб получают многие юристы – по данным за 2008 год, она составила 11,2 миллиона долларов¹ – тогда как экономисты, причастные к патентам на программное обеспечение, не способны извлечь из них ни цента прибыли.

¹ См. esp.wikidot.com/local--files/2008-state-of-softpatents/feb_08-summary_report.pdf.

У разработчиков, работающих за пределами США, тоже не очень много поводов для радости. Некоторым из них уже пришлось отозвать свое программное обеспечение с рынка США, чтобы избежать рэкета со стороны патентных троллей. Это означает, что некоторые инновационные программы более недоступны на территории США, что может дать конкурентные преимущества неамериканским компаниям. Кроме того, антипиратское торговое соглашение (Anti-Counterfeiting Trade Agreement, АСТА) охватывает патенты всех видов и уже принято многими странами во всем мире, включая Евросоюз, кроме Бразилии, Китая и России. Кроме того, так называемая Европейская «единая патентная система» (www.unitary-patent.eu), вероятнее всего, приобретет черты американской патентной системы и распространится на весь Евросоюз.

Программное обеспечение является одной из форм интеллектуальной собственности, которая прекрасно защищена авторским правом. (Например, Билл Гейтс (Bill Gates) некоторое время стал самым богатым человеком исключительно за счет авторского права и задолго до появления пагубной идеи патентовать программное обеспечение.) Несмотря на успех распространения авторского права на программное обеспечение, США и многие другие страны приняли или были вынуждены принять международные торговые соглашения по включению программного обеспечения в свои патентные законодательства. Представьте на мгновение, что все ничтожные патенты (то есть очевидные, допускающие слишком широкое толкование или для которых существуют известные технические решения, выработанные до подачи заявки на регистрацию) неожиданно исчезли. Это привело бы к резкому снижению фактов вымогательства и создало бы плодородную почву для инноваций. Но при этом остается открытым самый важный вопрос: является ли программное обеспечение предметом патентного права вообще?

В большинстве стран, включая США, невозможно запатентовать математическую формулу, даже новую и оригинальную. Даже при том, что математические формулы являются идеями (которые призваны «защищать» патенты). А из тезиса Черча-Тьюринга известно, что логику любого программного обеспечения можно разложить на математические формулы, то есть в действительности программное обеспечение является набором математических формул, записанных в специфическом виде. Именно этот аргумент приводит Дональд Кнут (Donald Knuth) против патентов на программное обеспечение. (См. краткую и увлекательную статью профессора Кнута на эту тему: www.progfree.org/Patents/knuth-to-ptto.txt.)



Проблема разрешима, но требует законодательного запрещения патентов на программное обеспечение (или приравнивания программного обеспечения к математике, каковым оно и является) и ограничения господства патентных бюро (чи доходы часто пропорциональны количеству выдаваемых ими патентов, независимо от их важности). Это непростая задача, потому что привлечение внимания политиков требует денег, а те, кто может позволить себе приобретать патенты, могут лоббировать нужные законы. Кроме того, эта тема настолько бесперспективна в смысле прибыли и настолько узкоспециальна, что сделать на ней политическую карьеру едва ли возможно. Но есть люди, ратующие за изменения в патентном законодательстве и предпринимающие попытки что-то сделать на двухпартийной основе. Пожалуй, лучшей отправной точкой, где можно узнать больше, почему патенты на программное обеспечение так разрушительны и как бороться с ними, является сайт endsoft-patents.org (и www.nosoftwarepatents.com – в Европе).



С. Список литературы

«Advanced Programming in the UNIX® Environment, Second Edition» W. Richard Stevens and Stephen A. Rago (Addison-Wesley, 2005, ISBN-13: 978-0-201-43307-4)¹

Всестороннее введение в программирование для операционной системы Unix с использованием системных вызовов Unix и стандартной библиотеки языка C. (Примеры в книге написаны на языке C.)

«The Art of Multiprocessor Programming» Maurice Herlihy and Nir Shavit (Morgan Kaufmann, 2008, ISBN-13: 978-0-12-370591-4)

Эта книга является исчерпывающим введением в основном в низкоуровневое программирование многопоточных приложений. Включает небольшие, но действующие примеры (на Java), демонстрирующие все ключевые приемы.

«Clean Code: A Handbook of Agile Software Craftsmanship» Robert C. Martin (Prentice Hall, 2009, ISBN-13: 978-0-13-235088-4)²

Эта книга описывает способы решения множества «тактических» проблем, возникающих в программировании: выбор хороших имен, проектирование функций, рефакторинг кода и др. В книге приводится множество интересных и нужных идей, способных помочь программистам улучшить свой стиль программирования и сделать их программы более простыми в сопровождении. (Книга содержит примеры на языке Java.)

¹ Ричард Стивенс У., Стивен Раго. UNIX. Профессиональное программирование». – Символ-Плюс, 2007, ISBN: 5-93286-089-8. – *Прим. перев.*

² Роберт Мартин. Чистый код. Создание, анализ и рефакторинг. Библиотека программиста. – Питер, 2010, ISBN: 978-5-49807-381-1. – *Прим. перев.*

«Code Complete: A Practical Handbook of Software Construction, Second Edition» Steve McConnell (Microsoft Press, 2004, ISBN-13: 978-0-7356-1967-8)¹

Эта книга рассказывает, как создавать надежное программное обеспечение, выходя за рамки специфики языка в область идей, принципов и приемов. Книга наполнена разнообразными идеями, способными заставить любого программиста глубже задуматься о программировании.

«Design Patterns: Elements of Reusable Object-Oriented Software» Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995, ISBN-13: 978-0-201-63361-0)²

Одна из самых значительных книг современности о программировании, даже при том, что является не самым легким чтением. Шаблоны проектирования являются увлекательной темой и имеют огромное практическое значение в обычном программировании.

«Domain-Driven Design: Tackling Complexity in the Heart of Software» Eric Evans (Addison-Wesley, 2004, ISBN-13: 978-0-321-12521-7)³

Очень интересная книга о проектировании программного обеспечения, особенно полезна будет разработчикам, работающим в составе коллективов. В этой книге речь идет о создании и совершенствовании предметных моделей, представляющих цель проектирования системы, и о создании универсального языка, посредством которого все, не только программисты, причастные к системе смогут обмениваться идеями.

¹ Стив Макконнелл. Совершенный код: Практическое руководство по разработке программного обеспечения. – Питер, 2005, ISBN: 5-469-00822-3. – *Прим. перев.*

² Эрих Гамма, Ричард Хелм, Ральф Джонсон и Джон Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – Питер, 2001, ISBN: 5-272-00355-1. – *Прим. перев.*

³ Эрик Эванс. Предметно-ориентированное проектирование (DDD). Структуризация сложных программных систем. – Питер, 2010, ISBN: 978-5-8459-1597-9. – *Прим. перев.*

«Don't Make Me Think!: A Common Sense Approach to Web Usability, Second Edition» Steve Krug (New Riders, 2006, ISBN-13: 978-0-321-34475-5)¹

Короткая, интересная и очень практичная книга об особенностях проектирования удобного веб-интерфейса, основанная на обширных исследованиях и опыте. Применение простых и понятных идей, излагаемых в этой книге, позволит повысить качество веб-сайта любого размера.

«Linux Programming by Example: The Fundamentals» Arnold Robbins (Prentice Hall, 2004, ISBN-13: 978-0-13-142964-2)²

Интересное и доступное введение в программирование для операционной системы Linux с использованием интерфейса системных вызовов. (Книга содержит примеры на языке C.)

«Mastering Regular Expressions, Third Edition» Jeffrey E.F. Friedl (O'Reilly, 2006, ISBN-13: 978-0-596-52812-6)³

Ставшее стандартом описание регулярных выражений – очень интересная и практичная книга.

¹ Стив Круг. Веб-дизайн: книга Стива Круга, или «Не заставляйте меня думать!». – 2-е изд. – Символ-Плюс, 2008, ISBN: 5-93286-099-5. – *Прим. перев.*

² Арнольд Роббинс. Linux: программирование в примерах. – 3-е изд. – Кудиц-Пресс, 2008, ISBN: 978-5-91136-056-6. – *Прим. перев.*

³ Джеффри Фридл. Регулярные выражения. – 3-е изд. – Символ-Плюс, 2008, ISBN: 5-93286-121-5. – *Прим. перев.*



Предметный указатель

А

Абстрактные и конкретные типы, 39
Агрегирование, 323, 349
Анонимные
 структуры, 348
 функции, 61, 146, 151, 267, 271,
 278, 290, 306, 367
Аргументы
 командной строки, 34, 35
 функций, 283
Асинхронные каналы, 265
Атомарное обновление, 427
Аудиоформат Vorbis Audio, 174

Б

Быстрая конкатенация строк, 122

В

Веб-приложения, 103
Веб-сайты
 code.google.com, 524
 endsoftpatents.org, 552
 godashboard.appspot.com, 512
 golang.org, 20, 546
 nosoftwarepatents.com, 552
 qtrac.eu, 12
Ветвление, 247
Взаимно рекурсивные функции, 291
Взаимоблокировка, 400
Виртуальные функции, 324
Возврат каретки (`\r`), 116

Возвраты в регулярных
 выражениях, 161
Возвращаемые значения, 43, 55, 64,
 242
 именованные, 58, 272, 282, 389
 неименованные, 282
Встраивание, 323, 332, 343, 349, 371,
 378
Встроенные типы
 bool, 357
 error, 359
 float32, 359
 float64, 359
Встроенные функции
 append(), 44, 46, 82, 109, 197, 198,
 205, 207, 208, 209, 232, 240, 316,
 318, 471, 494, 531
 cap(), 43, 196, 206, 240
 close(), 70, 240, 271, 433
 complex(), 100, 240
 copy(), 206, 240, 341
 delete(), 215, 240
 imag(), 240
 len(), 31, 43, 118, 196, 200, 201,
 215, 232, 240
 make(), 45, 69, 197, 198, 216, 240,
 265, 494
 new(), 191, 192, 199, 240, 437
 panic(), 53, 100, 240, 246, 252, 273,
 282
 real(), 240
 recover(), 53, 100, 240, 273, 280,
 371

Выбор

по значению выражения, 250

по типу, 253, 356, 361

Вызов функций, 283**Выравнивание**

по левому краю, 132, 140

по правому краю, 132, 140

Высшего порядка

функции, 305

Г**Группировка констант, инструкций**

импортирования и объявлений

переменных, 79

Д

Двоичные файлы, 488

Двунаправленные каналы, 265

Деление на ноль, 98

Динамическая (утиная)

типизация, 38, 54, 324, 340

Документирование пакетов, 516

Дополнение слева, 132

Доступ поочередный, 402, 423, 430

З

Забой (`\b`), 116

Завершение, 399

Замыкания, 64, 146, 213, 289, 445

Затенение переменных, 243, 246

Затеняющие переменные, 357

Значения, 184, 324

И**Идентификаторы**

в языке Go, 67, 76

пустые, 368

Изменение

отображений, 220

срезов, 204

Измерение эффективности, 522

Именованные

возвращаемые значения, 242, 272,

282, 389

ссылки, 162

в регулярных выражениях, 169

Именованные и неименованные

пользовательские типы, 39

Инвертирование отображений, 222

Инвертированные отображения, 233

Индексирование

срезов, 201

строк, 124

Инициализация, 47

переменных, 31, 105

срезов, 33

Инструкции

`break`, 42, 249, 261, 263, 419

`continue`, 177, 261, 263

`defer`, 52, 70, 271, 272, 279, 290

`else`, 249

`fallthrough`, 250, 252

`func`, 290

`go`, 264, 288, 290, 404, 405, 451

`goto`, 263

`if`, 31, 241, 243, 247, 300

`import`, 29, 514, 523

`package`, 512, 518

`return`, 42, 47, 56, 63, 93, 242, 243,

249, 277, 282

`select`, 69, 247, 261, 265, 267, 406,

419, 420, 451

`switch`, 146, 246, 247, 249, 257, 261,

300, 356, 361, 531

`type`, 518

ветвления, 247

необязательная, 253

необязательные, 247, 250

простые, 247, 250

цикл `for`, 36, 42, 63, 105, 122, 151,

177, 193, 202, 219, 224, 239, 242,

257, 259, 261, 404, 405, 418, 420,

428

Инструменты

- 5g, 6g, 8g, 21
- 5l, 6l, 8l, 21
- cgo, 22
- gc, 22
- gccgo, 22
- go build, 26, 514
- godoc, 517, 518, 526
- go fix, 525
- gofmt, 239, 526
- go get, 524
- go help, 525
- go install, 27, 514
- go test, 522
- go version, 24
- go vet, 525
- gonow (сторонний), 23
- gorun (сторонний), 23

Интерпретируемые строковые литералы, 115**Интерфейсы, 39, 325, 336, 372, 380, 402**

- пустые, 39

Истинные функции, 309**Итерации**

- по массивам, 261
- по отображениям, 221, 261
- по срезам, 202, 261
- по строкам, 261

К**Каналы, 68, 69, 264, 401**

- асинхронные, 265
- двунаправленные, 265
- однаправленные, 265
- синхронные, 265

Кен Томпсон (Ken Thompson), 12**Ключевые слова, 77**

- chan, 69, 70, 265, 401, 451
- const, 78, 79
- func, 30, 43, 281
- interface, 423
- iota, 80, 81

- nil, 46, 278, 328

- range, 36, 63, 105, 122, 151, 193, 202, 219, 257, 259, 261, 428

- struct, 67, 176, 349

- type, 255, 326

- var, 34, 78, 241

Кодировки

- 7-битная ASCII, 114

- с символами фиксированного и переменного размера, 114

Комментарии в языке Go, 29, 76**Конкатенация строк (быстрая), 122****Конкретные и абстрактные типы, 39****Конструкторы, 47****Копирование строк при записи, 184****Л****Линейный поиск, 212**

- в сравнении с методом половинного деления, 212

Литералы, 86

- вещественных чисел, 79

- комплексных чисел, 79, 100

- символов, 37

- составные, 34, 68, 71, 198, 216, 228
- строк, 115

- целочисленные, 79

Логическое выражение, 247**Локальные переменные, 64, 186****М**

- Максимальное число символов для вывода, 133, 140

Массивы, 184, 195

- изменяемые значения, 195
- итерации, 261
- многомерные, 195

Мемоизация истинных функций, 309**Метки, 262, 419****Методы, 325, 328, 351**

- Error(), 46, 51

Less(), 211
String(), 81, 82, 203, 218, 330, 336,
338
доступа, 335
множество, 245, 331
определение, 43
перегрузка, 329
переопределение, 333
Методы-выражения, 333
Минимальная ширина поля
вывода, 133, 140
Многомерные
массивы, 195
срезы, 33, 197, 262
Модульное тестирование, 520

Н

Наборы методов, 39
Наследование, 371, 378, 546
Научная форма записи, 138
Неизменяемость строк, 117
Неименованные
возвращаемые значения, 243, 282
структуры, 348
Неименованные и именованные
пользовательские типы, 39
Необязательная инструкция, 247,
250, 253
Необязательные аргументы, 286
Несвязанные методы (методы-
выражения), 334
Неэкспортируемые
идентификаторы, 78, 335
Нормализация
по пробельным символам, 150
Юникода, 119
Нотация O(...), 123
Нулевое значение, 79, 80, 198, 242,
328, 335, 391
Нумерованные ссылки в регулярных
выражениях, 168

О

Область видимости, 186
Обмен значениями, 241
Обобщенные функции, 298
Обработка ошибок, 53, 273
Обратный слеш (\), 116
Объявления, порядок
следования, 38
Однонаправленные каналы, 265
Операторы
арифметические, применимые
только к встроенным
целочисленным типам, 88
перегрузка, 90
, выбора, 194, 349
--, декремента, 36, 239, 241
++, декремента, 86
, деления, 86
, деления по модулю,
спецификатор формата, 88
%=: деления по модулю
с присваиванием, 88
+=, добавления, 37, 86, 117, 118,
184
[], индексирования и извлечения
среза, 48, 118, 125, 198
++, инкремента, 36, 86, 239, 241
&&, логическое И, 83, 84
||, логическое ИЛИ, 83, 84
!, логическое НЕ, 84
..., многоточия, 196, 204, 281, 284
!=, неравенства, 214
&, получения адреса и
поразрядное И, 73, 143, 186, 218
<-, посылки в канал / приема
из канала, 401
<-, приема/передачи, 265
=, присваивания, 34, 241
!=, проверки на неравенство, 83,
84, 98, 100
==, проверки на равенство, 83, 84,
98, 100
==, равенства, 214

- + , сложения, конкатенации
 - и унарный плюс, 37, 117, 118
- := , сокращенного объявления
 - переменных, 30, 31, 34, 78, 241, 242
- > , сравнения, 83, 84
- = , уменьшения, 86
- / , умножения/деления, 86
- * , умножения, разыменования
 - и объявления указателя, 45, 86, 187, 188
- <- , чтения/записи в канал, 71
- Определение методов, 43
- Отладка, 141
- Отображения, 214
 - изменение, 220
 - инвертирование, 222
 - инвертированные, 233
 - итерации, 221, 261
 - поиск, 219

П

- Пакеты, 29, 512
 - main, 29
 - для работы с коллекциями, 529
 - документирование, 516
 - пользовательские, 512
 - псевдонимы для имен
 - пакетов, 523
 - сторонние, 524
- Параметры, 40, 186, 324
 - функций, 283
- Патенты на программное
 - обеспечение, 548
- Перевод
 - строки (\n), 76, 116
 - формата (\f), 116
- Перегрузка
 - методов, 329
 - операторов, 90
- Переключатель типа, 254

- Переменные, 185, 339, 367
 - затенение, 243, 246
 - затеняющие, 357
 - инициализация, 105
 - локальные, 64, 186
 - окружения, 24, 513
 - GOPATH, 27, 28, 40, 513
 - GOROOT, 21, 24, 25, 40, 513
 - PATH, 25
 - оператор сокращенного объявления (:=), 78
 - сокращенное объявление, 241, 242
- Переопределение методов, 333
- Перечисления, 80
- Платформеннозависимый
 - программный код, 515
- Подстановка имен файлов (globbing), 229, 515
- Поиск
 - в отображениях, 219
 - в срезах, 212
 - методом половинного деления, 212
 - в сравнении с линейным
 - поиском, 212
 - строк, 120
- Полиморфизм, 254
- Пользовательские
 - пакеты, 512
 - типы, 141, 326
 - именованные и
 - неименованные, 39
 - функции, 281
- Поочередный доступ, 402, 423, 430
- Порядок
 - объявлений, 38
 - следования байтов, 115, 490, 495
- Предопределенные
 - идентификаторы, 77
- Преобразования, 89, 243, 364
 - типа int в тип float, 64, 89
 - типа rune в тип string, 121
 - типов, 243

Приведение типов, 245, 403
Приемник, 43, 49, 329, 332, 338, 351
Примеры, 23
 americanise, 50
 apachereport, 430
 bigdigits, 32
 cgrep, 413
 cgrep3, 516
 commandLineFiles(), 229, 516
 commas(), 450
 EqualFloat(), 98
 EqualFloatPrec(), 98
 filter, 407
 Filter(), 307
 findduplicates, 441
 FuzzyBool, 357
 guess_separator, 223
 hello, 28
 Humanize(), 137
 indent_sort, 312
 IndexSlicer(), 303
 InsertStringSlice(), 207
 InsertStringSliceCopy(), 205
 invoicedata, 456
 logPanics(), 280
 m3u2pls, 173
 Memoize(), 309
 omap, 382, 518, 520
 pack, 499
 Pad(), 136
 pi_by_digits, 90
 polar2cartesian, 66
 RemoveStringSlice(), 209
 RemoveStringSliceCopy(), 208
 RomanForDecimal(), 312
 safemap, 422
 shaper, 365
 SimplifyWhitespace(), 150
 SliceIndex(), 305
 SortFoldedStrings(), 211
 stacker, 40, 513
 statistics, 103

 statistics_nonstop, 278
 unpack, 499
 wordfrequency, 227
 сортировка с учетом отступов, 312
Пробельные символы, 127, 150
Проверка, 334
Простая инструкция, 247, 250
Процедурное
 программирование, 281
Псевдонимы для имен пакетов, 523
Пустая структура, 415
Пустые
 значения, 55
 идентификаторы (_), 59, 78, 202,
 222, 241, 368, 524
Путь к импортируемым пакетам, 41

P

Пасс Кокс (Russ Cox), 161, 546
Расширение имени файла, 249
Рекурсивные функции, 291, 388
Рекурсия, 291, 388
 хвостовая, 295
Роберт Гризмер (Robert
 Griesemer), 12

C

Сборка программ на языке Go, 25
Сборщик мусора, 64, 183, 186
Сигнал (\a), 116
Символы 116
 перевода строки, 76
 " кавычки, 115
 [], квадратные скобки, 239
 (), круглые скобки, 239
 &&, логический оператор И, 83, 84
 ||, логический оператор ИЛИ, 83,
 84
 !, логический оператор НЕ, 84
 /* ... */, многострочные
 комментарии, 29, 76

- \, обратный слеш, 116
- //, однострочные
 - комментарии, 29, 76
- _, пустой идентификатор, 59, 78, 202, 222, 241, 368, 524
- ;; точка с запятой, 238
- {}, фигурные скобки, 30, 239
- Символьные литералы, 37
- Синхронизация, 70
- Синхронные каналы, 265
- Сокращенный порядок вычислений, 83
- Сортировка
 - срез, 209
 - строка, 120
 - с учетом отступов (пример), 312
- Составные литералы, 34, 68, 71, 198, 216, 228
- Специальные функции
 - init(), 68, 276, 287, 367, 523
 - main(), 29, 263, 276, 287
- Спецификаторы формата (пакет fmt), 133
- Спецификация языка Go, 99
- Сравнение, 83
 - значений типа float, 64, 98
 - строка, 117
- Срезы, 33, 184, 196, 300
 - емкость, 198
 - извлечение из среза, 201
 - изменение, 204
 - индексирование, 201
 - итерации, 202, 261
 - многомерные, 33, 197, 262
 - поиск, 212
 - сортировка, 209
 - строка, 124
 - форматирование, 139
- Ссылки, 45, 184, 192, 214, 402
 - на функции, 128, 195
- Стандартная библиотека, 526

- Сторонние пакеты, 524
- Строки, 184
 - извлечение срезов, 124
 - индексирование, 124
 - интерпретируемые литералы, 115
 - итерации, 261
 - копирование при записи, 184
 - литералы, 115
 - неизменяемые значения, 117
 - поиск, 120
 - сортировка, 120
 - сравнение, 117
- Структуры анонимные, 348
- Сценарии на языке Go, 23

T

- Тарболлы, 502
- Теги структур, 353, 467, 468
- Типизация
 - динамическая (утиная), 38
 - строгая, 41
- Типы
 - множество методов, 245, 331
 - набор методов, 39
 - пользовательские, 326
 - преобразование, 243
 - приведение, 245
- Точка входа, 29
- Точность представления
 - вещественных чисел, 97
- Трассировочная информация, 274

У

- Указатели, 45, 49, 183, 186, 218, 331, 338, 360, 402, 456
 - форматирование, 143
- Ука() (пакет math), 94
- Упражнения
 - archive_file_list, 319
 - CommonPathPrefix(), 320

CommonPrefix(), 320
Flatten(), 235
font, 392
imagetag, 452
IsPalindrome(), 319
linkcheck, 543
linkutil, 542
Make2D(), 236
oslice, 395
ParseIni(), 237
PrintIni(), 237
quadratic, 111
safeslice, 451
shaper, 393
sizeimages, 453
statistics, 111
UniqueInts(), 235
unpack, 509
utf16-to-utf8/utf16-to-utf8, 510
Установка Go, 21

Ф

Фабричные функции, 291, 357, 377

Файлы

.go, 513
.gz, 499
.jpeg и .jpg, 371
.m3u, 173
.pls, 174
.png, 371

Фигурные скобки, 30

Форматирование

байтов, 140
вещественных значений, 137
для отладки, 141
комплексных значений, 137
логических значений, 134
отображений, 144
символов, 136
срезов, 139

строк, 139
указателей, 143
целочисленных значений, 134
Функции, 281
анонимные, 61, 146, 151, 267, 271,
278, 290, 306, 367
аргументы, 283
взаимно рекурсивные, 291
виртуальные, 324
выбор во время выполнения, 295
вызов, 283
высшего порядка, 305, 327
замыкания, 64, 146, 213, 289
истинные, 309
конструкторы, 47
memoизация, 309
необязательные аргументы, 286
обертки, 280, 290
обобщенные, 298
параметры, 283, 324
пользовательские, 281
рекурсивные, 291, 388
с переменным числом
аргументов, 254, 281, 284
ссылки, 128
фабричные, 291, 357, 377
Функции-конструкторы, 335

Х

Хвостовая рекурсия, 295

Ц

Целочисленные литералы, 79

Ч

Чарльз Энтони Ричард Хоар
(C. A. R. Hoare), 14
Числа, форматирование, 134

Э

Экранированные
 последовательности, 116, 472, 475
Экспоненциальная форма
 записи, 96, 138
Экспортируемые
 идентификаторы, 77, 259, 335
Эндрю Джерранд (Andrew
 Gerrand), 266

А

Abs() (пакет cmplx), 102
Abs() (пакет math), 94
Acosh() (пакет cmplx), 102
Acosh() (пакет math), 94
Acos() (пакет cmplx), 102
Acos() (пакет math), 94
Add(), метод
 типа Int, 92
Add() (тип WaitGroup), 443, 445, 447
After() (пакет time), 420, 536
americanise, пример, 50
arachereport, пример, 430
AppendBool() (пакет strconv), 153
AppendFloat() (пакет strconv), 153
AppendInt() (пакет strconv), 153
AppendQuoteRuneToASCII() (пакет
 strconv), 154
AppendQuoteRune() (пакет
 strconv), 153
AppendQuoteToASCII() (пакет
 strconv), 154
AppendQuote() (пакет strconv), 153
AppendUInt() (пакет strconv), 154
append(), встроенная функция, 44,
 46, 197, 198, 205, 207, 208, 209, 232,
 240, 471, 494, 531

Ю

Юникод, 77, 114, 119
 нормализация, 119
 пробельные символы, 127

Я

Ян Ланс Тейлор (Ian Lance
 Taylor), 12

append(), функция (встроенная), 82,
 109, 316, 318
archive_file_list, упражнение, 319
Args, срез (пакет os), 31, 33, 35
ASCII, кодировка символов, 114
Asinh() (пакет cmplx), 102
Asinh() (пакет math), 94
Asin() (пакет cmplx), 102
Asin() (пакет math), 94
Atan2() (пакет math), 94
Atanh() (пакет cmplx), 102
Atanh() (пакет math), 94
Atan() (пакет cmplx), 102
Atan() (пакет math), 94
Atoi() (пакет strconv), 154, 156, 179,
 479

В

base64, пакет (пакет encoding), 534
Base() (пакет filepath), 35, 176, 369
ВАТ-файл. См. Пакетный файл
bigdigits, пример, 32
BigEndian, переменная (пакет
 binary), 490
big, пакет
 Int, тип, 89
 Rat, тип, 60

big, пакет (пакет math), 534
ProbablyPrime(), 534
binary, пакет (пакет encoding), 489,
492, 534
BigEndian, переменная, 490
LittleEndian, переменная, 490
Read(), 495, 497
Write(), 489, 495
Bool() (тип Value), 538
bool (тип, встроенный), 83, 247, 251,
357, 402
форматирование, 134
break, инструкция, 42, 249, 261, 263,
419
BSON, формат, 467
Buffer, тип (пакет bytes), 122, 150, 257
ReadRune(), 152
String(), 122
WriteString(), 122, 257
bufio, пакет, 50, 57, 62
NewReader(), 58, 229
NewWriter(), 58
Reader, тип, 229, 484
bytes, пакет, 527
Buffer, тип, 122, 150, 257
TrimRight(), 422
byte (тип встроенный), 36, 87, 142, 244
форматирование, 140

C

Caller() (пакет runtime), 368
Call() (тип Value), 539
CanBackquote() (пакет strconv), 154
cap(), встроенная функция, 43, 196,
206, 240
Cbrt() (пакет math), 94
Ceil() (пакет math), 94
cgrep3, пример, 516
cgrep, пример, 413
chan, ключевое слово, 69, 70, 265,
401, 451
close(), встроенная функция, 70, 240,
433

Close() (тип File), 52, 500
close(), функция (встроенная), 271
CMD-файл. См. Пакетный файл
cmplx, пакет (пакет math), 100, 535
Abs(), 102
Acos(), 102
Acosh(), 102
Asin(), 102
Asinh(), 102
Atan(), 102
Atanh(), 102
Conj(), 102
Cos(), 102
Cosh(), 102
Cot(), 102
Exp(), 102
Inf(), 102
IsInf(), 102
IsNaN(), 102
Log(), 102
Log10(), 102
NaN(), 102
Phase(), 102
Polar(), 102
Pow(), 102
Rect(), 102
Sin(), 102
Sinh(), 102
Sqrt(), 102
Tan(), 102
Tanh(), 102
code.google.com (веб-сайт), 524
commandLineFiles(), пример, 229,
516
commas(), пример, 450
CommonPathPrefix(),
упражнение, 320
CommonPrefix(), упражнение, 320
CompilePOSIX() (пакет regexp), 162
Compile() (пакет regexp), 60, 162,
275
complex64 (тип встроенный), 93, 101,
244

complex128 (тип встроенный), 93,
101, 244
 литералы, 79
complex(), встроенная функция, 240
Complex() (тип Value), 538
complex(), встроенная функция, 100
Conj() (пакет cmplx), 102
const, ключевое слово, 78, 79
container, пакет, 529
 heap, пакет, 529
 list, пакет, 530
 ring, пакет, 531
Contains() (пакет strings), 147
continue, инструкция, 177, 261, 263
Coprysign() (пакет math), 94
copy(), встроенная функция, 206,
240, 341
Copy() (пакет io), 447, 501
Cosh() (пакет cmplx), 102
Cosh() (пакет math), 94
Cos() (пакет cmplx), 102
Cos() (пакет math), 94
Cot() (пакет cmplx), 102
Count() (пакет strings), 147, 225
Create() (пакет os), 370
Create() (тип File), 52
crypto, пакет, 535
 rand, пакет, 536
 sha512, пакет, 535
CSP (Communicating Sequential
Processes – взаимодействующие
последовательные процессы),
модель, 398
csv, пакет (пакет encoding), 533

D

database, пакет
 sql, пакет, 531
Decode()
 пакет gob, тип Decoder, 487
 пакет json, тип Decoder, 465, 466
 пакет xml, тип Decoder, 473, 475
DecodeConfig() (пакет image), 452

DecodeLastRuneInString() (пакет
utf), 295
DecodeLastRuneInString() (пакет
utf8), 126, 159
DecodeLastRune() (пакет utf8), 159
DecodeRuneInString() (пакет utf), 295
DecodeRuneInString() (пакет
utf8), 126, 159
DecodeRune() (пакет utf8), 159
Decoder, тип
 пакет gob, тип Decode(), 487
 пакет json, тип Decode(), 465, 466
 пакет xml, тип Decode(), 473, 475
DeepEqual() (пакет reflect), 302, 537
defer, инструкция, 52, 70, 271, 272,
279, 290
delete(), встроенная функция, 215,
240
Dim() (пакет math), 94
Done() (тип WaitGroup), 443, 445, 447
DrawMask() (пакет draw), 367
draw, пакет (интерфейс Image), 367
Draw() (пакет draw), 367
draw, пакет (пакет image)
 Draw(), 367
 DrawMask(), 367
 Image, интерфейс, 403
Duration, тип (пакет time), 420

E

Elem() (тип Value), 539
else, инструкция, 249
Encode()
 пакет gob, тип Encoder, 485
 пакет json, тип Encoder, 462, 466
 пакет xml, тип Encoder, 470, 472
EncodeRune() (пакет utf8), 159
Encoder, тип
 пакет gob, тип Encode(), 485
 пакет json, 462
 Encode(), 462, 466
 пакет xml, тип Encode(), 470, 472
Encode() (пакет jpeg), 371

encoding, пакет, 534
 base64, пакет, 534
 binary, пакет, 534
 csv, пакет, 533
endsoftpatents.org, веб-сайт, 552
EOF, константа (пакет io), 422, 480,
 490, 507
EOF (пакет io), 56, 60
EqualFloatPrec(), пример, 98
EqualFloat(), пример, 98
EqualFold() (пакет strings), 147, 213
Erfc() (пакет math), 94
Errorf() (пакет fmt), 55, 85, 133
errors, пакет, 42
 New(), 46, 55
Error(), метод, 46, 51
error (тип встроенный), 42, 275, 359
EscapeString() (пакет html), 110
exec, пакет (пакет os), 535
Exit() (пакет os), 35, 37, 53, 176
Exp2() (пакет math), 94
ExpandString() (тип Regexp), 166
Expand() (тип Regexp), 166
Expn1() (пакет math), 94
Exp(), метод
 типа Int, 92
Exp() (пакет cmplx), 102
Exp() (пакет math), 94
Ext() (пакет filepath), 411
E, константа (пакет math), 94

F

fallthrough, инструкция, 250, 252
Fatal() (пакет log), 37, 53, 176
Fatalf() (пакет log), 37
FieldsFunc() (пакет strings), 146,
 147, 231
Fields() (пакет fmt), 479
Fields() (пакет strings), 63, 109, 147,
 150
FileInfoHeader() (пакет zip), 501
FileInfo, интерфейс (пакет os), 444,
 455, 501

FileInfo() (пакет zip), 506
FileInfo, тип (пакет os), 448, 499
filepath, пакет
 Base(), 176, 369
 FromSlash(), 180
 Separator, константа, 180
filepath, пакет (пакет path), 33, 533
 Base(), 35
 Ext(), 411
 Glob(), 515
 SkipDir, константа, 444
 Walk(), 442, 445
File, тип
 zip, пакет
 Mode(), 506
 ModTime(), 506
 Open(), 506
File, тип (пакет os)
 Close(), 52, 500
 Create(), 52
 Open(), 52
 OpenFile(), 52
 ReadAt(), 499
 Readdir(), 455
 Readdirnames(), 455
 Seek(), 499
 Stat(), 499, 501
 WriteAt(), 499
File, тип (пакет zip), 505
filter, пример, 407
Filter(), пример, 307
FindAllIndex() (тип Regexp), 166
FindAllStringIndex() (тип
 Regexp), 166
FindAllStringSubmatchIndex() (тип
 Regexp), 166
FindAllStringSubmatch() (тип
 Regexp), 166
FindAllString() (тип Regexp), 166
FindAllSubmatchIndex() (тип
 Regexp), 166
FindAllSubmatch() (тип Regexp), 166
FindAll() (тип Regexp), 166

- findduplicates, пример, 441
 - FindIndex() (тип Regexp), 166
 - FindReaderIndex() (тип Regexp), 167
 - FindReaderSubmatchIndex() (тип Regexp), 167
 - FindStringIndex() (тип Regexp), 167
 - FindStringSubmatchIndex() (тип Regexp), 167
 - FindStringSubmatch() (тип Regexp), 434
 - FindString() (тип Regexp), 167
 - FindSubmatchIndex() (тип Regexp), 167
 - FindSubmatch() (тип Regexp), 167
 - Find() (тип Regexp), 166
 - flag, пакет, 408, 535
 - Flatten(), упражнение, 235
 - Float32bits() (пакет math), 94
 - Float32frombits() (пакет math), 94
 - float32 (тип встроенный), 93, 359
 - Float64bits() (пакет math), 94
 - Float64frombits() (пакет math), 94
 - Float64sAreSorted() (пакет sort), 210
 - Float64s() (пакет sort), 104, 210
 - float64, тип (встроенный), 90, 93, 99, 137, 359, 402, 493
 - литералы, 79
 - преобразование из типа int, 89
 - сравнение значений, 98
 - форматирование, 137
 - Float() (тип Value), 538
 - Floor() (пакет math), 95
 - Flush() (интерфейс Writer), 58
 - fmt, пакет, 81, 128, 246, 527
 - Errorf(), 55, 85, 133
 - Fields(), 479
 - Fprint(), 78
 - Fprintf(), 133, 257, 476
 - Fscan(), 483
 - Fscanf(), 483, 484
 - Fscanln(), 483
 - Print(), 142
 - Printf(), 35, 55, 73, 133
 - Println(), 38, 101, 217
 - Scan(), 483
 - Scanf(), 483
 - Scanln(), 483
 - Split(), 479
 - Sprintf(), 232
 - Sprintf(), 82, 98, 133, 138, 139
 - Sscanf(), 73, 483
 - Sscanln(), 483
 - Stringer, интерфейс, 336, 338, 361
 - спецификаторы формата, 133
 - font, упражнение, 392
 - FormatBool() (пакет strconv), 154, 157, 478
 - FormatFloat() (пакет strconv), 154
 - FormatInt() (пакет strconv), 154, 158
 - FormatUInt() (пакет strconv), 154
 - Format() (тип Time), 464, 478, 492
 - for, цикл, 36, 42, 63, 122, 105, 151, 177, 193, 202, 219, 224, 239, 242, 257, 259, 261, 404, 405, 418, 420, 428
 - Fprint() (пакет fmt), 78
 - Fprintf() (пакет fmt), 133, 257, 476
 - Frexp() (пакет math), 95
 - FromSlash() (пакет filepath), 180
 - Fscan() (пакет fmt), 483
 - Fscanf() (пакет fmt), 483, 484
 - Fscanln() (пакет fmt), 483
 - FullRune() (пакет utf8), 159
 - FullRuneInString() (пакет utf8), 159
 - FuncForPC() (пакет runtime), 369
 - func
 - инструкция, 290
 - ключевое слово, 30, 43, 281
 - FuzzyBool, пример, 357
- ## G
- Gamma() (пакет math), 95
 - Glob() (пакет filepath), 515
 - GOARCH, константа (пакет runtime), 533
 - GobDecoder, интерфейс (пакет gob), 486

GobEncoder, интерфейс (пакет gob), 486
go build, инструмент, 26, 514
gob, пакет (пакет encoding), 485
 GobDecoder, интерфейс, 486
 GobEncoder, интерфейс, 486
godashboard.appspot.com (веб-сайт), 512
godoc, инструмент, 517
godoc, инструмент, 518, 526
go fix, инструмент, 525
gofmt, инструмент, 239, 526
go get, инструмент, 524
go help, инструмент, 525
go install, инструмент, 27, 514
golang.org, веб-сайт, 546
GOMAXPROCS() (пакет runtime), 414, 533
gonow, сторонний инструмент, 23
Google App Engine, 546
GOOS, константа (пакет runtime), 68, 533
GOPATH, переменная окружения, 27, 28, 40, 513
GOROOT, переменная окружения, 24, 25, 40, 513
gorun, сторонний инструмент, 23
go test, инструмент, 522
goto, инструкция, 263
go version, инструмент, 24
go vet, инструмент, 525
go, инструкция, 264, 288, 290, 404, 405, 451
go-подпрограммы (goroutines), 14, 65, 74, 263, 399
Go, язык программирования
 Dashboard, сайт, 12
 документация, 20
 идентификаторы, 67, 76
 история, 12
 кодировка символов в исходных текстах программ, 23
 комментарии, 29, 76

 спецификация, 99
 сценарии, 23
 установка, 21
guess_separator, пример, 223
gzip, пакет (пакет compress)
 NewReader(), 507
 NewWriter(), 503
 Reader, тип, 507

H

HandleFunc() (пакет http), 106, 281
HasPrefix() (пакет strings), 147, 330
HasSuffix() (пакет strings), 147, 176
Header, константа (пакет xml), 470
Header, тип (пакет tar), 504, 507
heap, пакет (пакет container), 529
hello, пример, 28
HTMLEscape() (пакет html), 110
html, пакет
 EscapeString(), 110
 HTMLEscape(), 110
http, пакет (пакет net), 536
 HandleFunc(), 106, 281
 ListenAndServe(), 106
 ResponseWriter, интерфейс, 106
Humanize(), пример, 137
Hypot() (пакет math), 95

I

IEEE-754, формат представления двоичных чисел, 93
if, инструкция, 31, 241, 243, 247, 300
Ilogb() (пакет math), 95
imagetag, упражнение, 452
Image, интерфейс
 пакет draw, 367, 403
 пакет image, 366, 403
image, пакет, 366, 534
 DecodeConfig(), 452
 Image, интерфейс, 366, 403
 NewRGBA(), 403
 ZP (нулевая точка), 367

- imag(), встроенная функция, 240
- import, инструкция, 29, 514, 523
- indent_sort, пример, 312
- IndexAny() (пакет strings), 147, 179
- IndexFunc() (пакет strings), 127, 128, 147
- IndexRune() (пакет strings), 147
- IndexSlicer(), пример, 303
- Index() (пакет strings), 147, 179
- Inf() (пакет cmplx), 102
- Inf() (пакет math), 95
- init(), специальная функция, 68, 276, 287, 367, 523
- InsertStringSliceCopy(), пример, 205
- InsertStringSlice(), пример, 207
- int8 (тип встроенный), 87, 492, 497
- int16 (тип встроенный), 87, 244
- int32 (тип встроенный), 87, 121, 142, 244, 492, 495
- int64 (тип встроенный), 87, 158, 277, 450
- Interface, интерфейс (пакет sort), 211, 314, 529
- interface, ключевое слово, 423
- interface{}, интерфейс, 197, 325, 337, 423, 530
- interface{}, тип, 39, 245, 246, 253, 255, 283, 299, 382
- Intn() (пакет rand), 536
- IntsAreSorted() (пакет sort), 210
- Ints() (пакет sort), 210
- int (тип встроенный), 81, 85, 87, 99, 142, 158, 241, 277, 303, 402, 495
 - литералы, 79
 - преобразование в тип float, 64, 89
 - преобразование в тип int, 64, 93
- форматирование, 134
- Int, тип (пакет big), 89
 - Add(), 92
 - Exp(), 92
 - Mul(), 92
- Int() (пакет rand), 536
- Int() (тип Value), 538
- invoicedata, пример, 456
- iota, ключевое слово, 80, 81
- ioutil, пакет (пакет io), 50, 62, 532
 - ReadAll(), 532
 - ReadFile(), 62, 65, 532
 - TempFile(), 532
 - WriteFile(), 533
- io, пакет, 50, 532
 - Copy(), 447, 501
 - EOF, 56, 60
 - EOF, константа, 422, 480, 490, 507
- ioutil, пакет, 532
 - Pipe(), 407
 - ReadCloser, интерфейс, 507, 508
 - Reader, интерфейс, 54, 56, 325, 340, 452, 460, 464, 473, 479, 495, 496, 508
 - ReadWriter, интерфейс, 54
 - WriteCloser, интерфейс, 503, 508
 - Writer, интерфейс, 54, 56, 129, 324, 447, 461, 462, 470, 485, 490, 508
- IsControl() (пакет unicode), 160
- IsDigit() (пакет unicode), 160
- IsGraphic() (пакет unicode), 160
- IsInf() (пакет cmplx), 102
- IsInf() (пакет math), 95
- IsLetter() (пакет unicode), 160
- IsLower() (пакет unicode), 160
- IsMark() (пакет unicode), 160
- IsNaN() (пакет cmplx), 102
- IsNaN() (пакет math), 95
- IsOneOf() (пакет unicode), 160
- IsPalindrome(), упражнение, 319
- IsPrint() (пакет unicode), 160
- IsPunct() (пакет unicode), 160
- IsSorted() (пакет sort), 210
- IsSpace() (пакет unicode), 128, 150, 160
- IsSymbol() (пакет unicode), 160
- IsTitle() (пакет unicode), 160
- IsUpper() (пакет unicode), 160
- Is() (пакет unicode), 160
- Itoa() (пакет strconv), 154, 158

J

J0() (пакет math), 95
J1() (пакет math), 95
Jn() (пакет math), 95
Join() (пакет strings), 31, 82, 147, 150
jpeg, пакет (пакет image), 371
 Encode(), 371
JSON (JavaScript Object Notation –
 форма записи объектов
 JavaScript), 458, 460
json, пакет (пакет encode)
 NewEncoder(), 462
json, пакет (пакет encoding), 461
 Marshal(), 463, 466
 Unmarshal(), 255, 466
 Unmarshaler, интерфейс, 466

L

LastIndex() (пакет strings), 147
LastIndexAny() (пакет strings), 147
LastIndexFunc() (пакет strings), 127,
 147
Ldexp() (пакет math), 95
len(), встроенная функция, 31, 43,
 196, 200, 201, 215, 232, 240
Len() (тип Value), 541
len(), встроенная функция, 118
Less(), метод, 211
Lgamma() (пакет math), 95
linkcheck, упражнение, 543
linkutil, упражнение, 542
ListenAndServe() (пакет http), 106
list, пакет (пакет container), 530
LiteralPrefix() (тип Regexp), 167
LittleEndian, переменная (пакет
 binary), 490
Ln2() (пакет math), 95
Ln10() (пакет math), 95
Lock() (тип RWMutex), 437
Log1p() (пакет math), 95
Log2E() (пакет math), 95
Log2() (пакет math), 95

Log10E() (пакет math), 95
Log10() (пакет cmplx), 102
Log10() (пакет math), 95
Logb() (пакет math), 95
logPanic(), пример, 280
log, пакет, 279, 535
 Fatal(), 37, 53, 176
 Fprintf(), 37
 SetFlags(), 535
 SetOutput(), 535
Log() (пакет cmplx), 102
Log() (пакет math), 95

M

m3u2pls, пример, 173
main, пакет, 29, 263, 276, 287
main() (специальная функция), 29,
 263, 276, 287
Make2D(), упражнение, 236
make(), встроенная функция, 45, 69,
 197, 198, 216, 240, 265, 494
Map() (пакет strings), 147, 151, 179,
 195
map (тип встроенный), 49, 62, 214,
 227
 быстрый поиск, 62
 форматирование, 144
Marshal() (пакет json), 463, 466
Match() (пакет regexp), 162
Match() (тип Regexp), 167, 422
MatchReader() (пакет regexp), 162
MatchReader() (тип Regexp), 167
MatchString() (пакет regexp), 162
MatchString() (тип Regexp), 167
math, пакет, 67, 99
 Abs(), 94
 Acos(), 94
 Acosh(), 94
 Asin(), 94
 Asinh(), 94
 Atan(), 94
 Atan2(), 94
 Atanh(), 94

- big, пакет, 534
- Cbrt(), 94
- Ceil(), 94
- cmplx, пакет, 535
- Copysign(), 94
- Cos(), 94
- Cosh(), 94
- Dim(), 94
- Erfc(), 94
- Exp(), 94
- Exp2(), 94
- Expm1(), 94
- E, константа, 94
- Float32bits(), 94
- Float32frombits(), 94
- Float64bits(), 94
- Float64frombits(), 94
- Floor(), 95
- Frexp(), 95
- Gamma(), 95
- Hypot(), 95
- llogb(), 95
- Inf(), 95
- IsInf(), 95
- IsNaN(), 95
- J0(), 95
- J1(), 95
- Jn(), 95
- Ldexp(), 95
- Lgamma(), 95
- Ln2(), 95
- Ln10(), 95
- Log(), 95
- Log1p(), 95
- Log2(), 95
- Log2E(), 95
- Log10(), 95
- Log10E(), 95
- Logb(), 95
- Max(), 95
- MaxInt32, константа, 99
- MaxUInt8, константа, 85
- Min(), 95
- MinInt32, константа, 99
- Mod(), 95
- Modf(), 96, 99, 138
- NaN(), 96
- Nextafter(), 96
- Phi, константа, 96
- Pi, константа, 96
- Pow(), 96
- Pow10(), 96
- rand, пакет, 536
- Remainder(), 96
- Signbit(), 96
- Sin(), 96
- SinCos(), 96
- Sinh(), 96
- Sqrt(), 96
- Sqrt2, константа, 96
- SqrtE, константа, 96
- SqrtPhi, константа, 96
- SqrtPi, константа, 96
- Tan(), 96
- Tanh(), 96
- Trunc(), 96
- Y0(), 96
- Y1(), 96
- Yn(), 96
- Ука(), 94
- Max() (пакет math), 95
- MaxInt32, константа (пакет math), 99
- MaxRune, константа (пакет unicode), 113
- MaxUInt8, константа (пакет math), 85
- Memoize(), пример, 309
- MethodByName() (тип Value), 541
- MinInt32, константа (пакет math), 99
- Min() (пакет math), 95
- MkdirAll() (пакет os), 505
- Mode(), пакет zip (тип File), 506

ModeType, константа (пакет os), 444
 Mod() (пакет math), 95
 Modf() (пакет math), 96, 99, 138
 ModTime(), пакет zip (тип File), 506
 Mul(), метод типа Int, 92
 MustCompilePOSIX() (пакет
 regex), 162
 MustCompile() (пакет regex), 60,
 162, 275
 Mutex, тип (пакет sync), 430

N

Name, тип (пакет xml), 468, 471
 NaN() (пакет cmplx), 102
 NaN() (пакет math), 96
 net, пакет, 536
 http, пакет, 536
 grpc, пакет, 537
 smtp, пакет, 537
 url, пакет, 537
 net, пакет (пакет http), 103
 new(), встроенная функция, 191,
 192, 199, 240, 437
 New() (пакет errors), 46, 55
 New() (пакет sha1), 447
 NewEncoder()
 json, пакет, 462
 NewReader() (пакет bufio), 58, 229
 NewReader() (пакет gzip), 507
 NewReader() (пакет strings), 147, 152
 NewReader() (пакет zip), 505
 NewReplacer() (пакет strings), 147
 NewRGBA() (пакет image), 403
 NewTicker() (пакет time), 536
 NewWriter() (пакет bufio), 58
 NewWriter() (пакет gzip), 503
 Nextafter() (пакет math), 96
 nil, ключевое слово, 46, 278, 328
 nosoftwarepatents.com, веб-сайт, 552
 NumCPU() (пакет runtime), 414, 533
 NumGoroutine() (пакет runtime), 446
 NumSubexp() (тип Regexp), 167

O

Ogg, контейнер, 174
 омар, пример, 382, 518, 520
 Open(), пакет zip (тип File), 506
 OpenFile() (тип File), 52
 Open() (пакет os), 452, 505
 Open() (тип File), 52
 O_RDWR, константа (пакет os), 499
 oslice, упражнение, 395
 os, пакет, 532
 Args, спрез, 31, 33, 35
 Create(), 370
 Exit(), 35, 37, 53, 176
 FileInfo, интерфейс, 444, 501
 FileInfo, тип, 448, 499
 File, тип, 54
 MkdirAll(), 505
 ModeType, константа, 444
 Open(), 452, 505
 O_RDWR, константа, 499
 Stderr, поток, 53, 176
 Stdin, поток, 52, 53
 Stdout, поток, 52, 53

P

package, инструкция, 512, 518
 pack, пример, 499
 Pad(), пример, 136
 panic(), встроенная функция, 53,
 100, 240, 246, 252, 273, 282
 Parse() (пакет time), 474, 479, 482
 ParseBool() (пакет strconv), 134, 155
 ParseFloat() (пакет strconv), 109,
 155, 156
 ParseIni(), упражнение, 237
 ParseInt() (пакет strconv), 155, 156
 ParseUint() (пакет strconv), 155, 157
 path, пакет, 533
 filepath, пакет, 533
 PATH, переменная окружения, 25
 Phase() (пакет cmplx), 102

Phi, константа (пакет math), 96
pi_by_digits, пример, 90
Pipe() (пакет io), 407
Pi, константа (пакет math), 96
png, пакет (пакет image), 371
polar2cartesian, пример, 66
Polar() (пакет cmplx), 102
Pow10() (пакет math), 96
Pow() (пакет cmplx), 102
Pow() (пакет math), 96
Printf() (пакет fmt), 35, 55, 73, 133
PrintIni(), упражнение, 237
Println() (пакет fmt), 38, 101, 217
Print() (пакет fmt), 142
ProbablyPrime() (пакет big), 534

Q

quadratic, упражнение, 111
QuoteMeta() (пакет regexp), 162
QuoteRuneToASCII() (пакет strconv), 156
QuoteRune() (пакет strconv), 155
QuoteToASCII() (пакет strconv), 156
Quote() (пакет strconv), 155, 158

R

rand, пакет
 crypto, пакет, 536
 Int(), 536
 Intn(), 536
 math, пакет, 536
range, ключевое слово, 36, 63, 105,
 122, 151, 193, 202, 219, 257, 259,
 261, 428
Rat, тип (пакет big), 90
ReadAll() (пакет ioutil), 532
ReadAt() (тип File), 499
ReadBytes() (тип Reader), 422
ReadCloser, интерфейс (пакет io), 507, 508
ReadCloser, тип (пакет zip), 505
Readdirnames() (тип File), 455

Readdir() (тип File), 455
Reader, интерфейс (пакет io), 54, 56,
 325, 340, 452, 460, 464, 473, 479,
 495, 496, 508
Reader, тип
 bufio, пакет
 ReadBytes(), 422
 ReadString(), 60
 паке strings, 152
Reader, тип (пакет bufio), 229, 484
Reader, тип (пакет gzip), 507
Reader, тип (пакет tar), 507
ReadFile() (пакет ioutil), 62, 65, 532
ReadRune() (тип Buffer), 152
ReadString() (тип Reader), 60
ReadWriter, интерфейс (пакет io), 54
Read() (пакет binary), 495, 497
real(), встроенная функция, 240
recover(), встроенная функция, 53,
 100, 240, 273, 280, 371
Rect() (пакет cmplx), 102
reflect, пакет, 245, 302, 537
 DeepEqual(), 302, 537
 TypeOf(), 537
 ValueOf(), 537
regexp, пакет, 145, 161, 275, 536
 Compile(), 60, 162, 275
 CompilePOSIX(), 162
 Match(), 162
 MatchReader(), 162
 MatchString(), 162
 MustCompile(), 60, 162, 275
 MustCompilePOSIX(), 162
 QuoteMeta(), 162
 Regexp, тип, 60, 421, 434
Regexp, тип
 ReplaceAllStringFunc(),
 функция, 61, 63
Regexp, тип (пакет regexp), 60, 402,
 421, 434
 Expand(), 166
 ExpandString(), 166
 Find(), 166

FindAll(), 166
 FindAllIndex(), 166
 FindAllString(), 166
 FindAllStringIndex(), 166
 FindAllStringSubmatch(), 166
 FindAllString-
 SubmatchIndex(), 166
 FindAllSubmatch(), 166
 FindAllSubmatchIndex(), 166
 FindIndex(), 166
 FindReaderIndex(), 167
 FindReaderSubmatchIndex(), 167
 FindString(), 167
 FindStringIndex(), 167
 FindStringSubmatch(), 167
 FindStringSubmatchIndex(), 167
 FindSubmatch(), 167
 FindSubmatchIndex(), 167
 LiteralPrefix(), 167
 Match(), 167, 422
 MatchReader(), 167
 MatchString(), 167
 NumSubexp(), 167
 ReplaceAll(), 162, 167
 ReplaceAllFunc(), 167
 ReplaceAllLiteral(), 168
 ReplaceAllLiteralString(), 168
 ReplaceAllString(), 168
 ReplaceAllStringFunc(), 168
 ReplaceAllStrings(), 162
 String(), 168
 SubexpNames(), 168
 Remainder() (пакет math), 96
 RemoveStringSlice(), пример, 209
 RemoveStringSliceCopy(),
 пример, 208
 Repeat() (пакет strings), 136
 Replace() (пакет strings), 109, 148,
 149
 ReplaceAll() (тип Regexp), 162, 167
 ReplaceAllFunc() (тип Regexp), 167
 ReplaceAllLiteralString() (тип
 Regexp), 168

ReplaceAllLiteral() (тип Regexp), 168
 ReplaceAllStringFunc() (тип
 Regexp), 61, 63, 168
 ReplaceAllStrings() (тип Regexp), 162
 ReplaceAllString() (тип Regexp), 168
 Request, тип (пакет http), 106
 ResponseWriter, интерфейс (пакет
 http), 106
 return, инструкция, 42, 47, 56, 63, 93,
 242, 243, 249, 277, 282
 ring, пакет (пакет container), 531
 RomanForDecimal(), пример, 312
 rpc, пакет (пакет net), 537
 RuneCountInString() (пакет
 utf8), 136, 159
 RuneCount() (пакет utf8), 159
 RuneLen() (пакет utf8), 159
 RuneStart() (пакет utf8), 159
 rune, тип (встроенный), 87, 114, 121,
 142, 244, 527
 преобразование в тип string, 121
 форматирование, 136
 runtime, пакет, 67, 533
 Caller(), 368
 FuncForPC(), 369
 GOARCH, константа, 533
 GOMAXPROCS(), 414, 533
 GOOS, константа, 68, 533
 NumCPU(), 414, 533
 NumGoroutine(), 446
 Version(), 533
 RWMutex, тип (пакет sync), 430
 Lock(), 437
 Unlock(), 437

S

safemap, пример, 422
 safeslice, упражнение, 451
 Scan() (пакет fmt), 483
 Scanf() (пакет fmt), 483
 Scanln() (пакет fmt), 483
 Search() (пакет sort), 209, 210, 212,
 213, 306

- SearchFloat64s() (пакет sort), 210
- SearchInts() (пакет sort), 210
- SearchStrings() (пакет sort), 210
- Seek() (тип File), 499
- select, инструкция, 69, 247, 261, 265, 267, 406, 419, 420, 451
- Separator, константа (пакет filepath), 180
- SetFlags() (пакет log), 535
- SetOutput() (пакет log), 535
- SHA-1? защищенный алгоритм хеширования, 441
- sha1, пакет (пакет crypto), New(), 447
- sha512, пакет (пакет crypto), 535
- shaper, пример, 365
- shaper, упражнение, 393
- Signbit() (пакет math), 96
- SimpleFold() (пакет unicode), 160
- SimplifyWhitespace(), пример, 150
- SinCos() (пакет math), 96
- Sinh() (пакет cmplx), 102
- Sinh() (пакет math), 96
- Sin() (пакет cmplx), 102
- Sin() (пакет math), 96
- sizeimages, упражнение, 453
- SkipDir, константа (пакет filepath), 444
- SliceIndex(), пример, 305
- smtp, пакет (пакет net), 537
- sort, пакет, 104, 209, 306, 536
 - Float64s(), 104, 210
 - Float64sAreSorted(), 210
 - Interface, интерфейс, 211, 314, 529
 - Ints(), 210
 - IntsAreSorted(), 210
 - IsSorted(), 210
 - Search(), 209, 210, 212, 213, 306
 - SearchFloat64s(), 210
 - SearchInts(), 210
 - SearchStrings(), 210
 - Sort(), 209, 210, 211, 314
 - Strings(), 210, 312
 - StringsAreSorted(), 210
 - Sort() (пакет sort), 209, 210, 211, 314
 - SortFoldedStrings(), пример, 211
 - Split() (пакет fmt), 479
 - Split() (пакет strings), 62, 63, 146, 148
 - SplitAfterN() (пакет strings), 146, 148
 - SplitAfter() (пакет strings), 146, 148
 - SplitN() (пакет strings), 146, 148
 - Sprint() (пакет fmt), 232
 - Sprintf() (пакет fmt), 82, 98, 133, 138, 139
 - sql, пакет (пакет database), 531
 - Sqrt() (пакет cmplx), 102
 - Sqrt() (пакет math), 96
 - Sqrt2, константа (пакет math), 96
 - SqrtE, константа (пакет math), 96
 - SqrtPhi, константа (пакет math), 96
 - SqrtPi, константа (пакет math), 96
 - Sscanf() (пакет fmt), 73, 483
 - Sscanln() (пакет fmt), 483
 - stacker, пример, 40, 513
 - statistics_nonstop, пример, 278
 - statistics, пример, 103
 - statistics, упражнение, 111
 - Stat() (тип File), 499, 501
 - Stderr, поток (пакет os), 53, 176
 - Stdin, поток (пакет os), 52, 53
 - Stdout, поток (пакет os), 52, 53
 - strconv, пакет, 145, 153, 527
 - AppendBool(), 153
 - AppendFloat(), 153
 - AppendInt(), 153
 - AppendQuote(), 153
 - AppendQuoteRune(), 153
 - AppendQuoteRuneToASCII(), 154
 - AppendQuoteToASCII(), 154
 - AppendUInt(), 154
 - Atoi(), 154, 156, 179, 479
 - CanBackquote(), 154
 - FormatBool(), 154, 157, 478
 - FormatFloat(), 154
 - FormatInt(), 154, 158
 - FormatUInt(), 154

- Itoa(), 154, 158
 - ParseBool(), 134, 155
 - ParseFloat(), 109, 155, 156
 - ParseInt(), 155, 156
 - ParseUInt(), 155, 157
 - Quote(), 155, 158
 - QuoteRune(), 155
 - QuoteRuneToASCII(), 156
 - QuoteToASCII(), 156
 - Unquote(), 156, 158
 - UnquoteChar(), 156
 - string (тип встроенный), 113, 184, 244
 - сравнение, 117
 - форматирование, 139
 - String(), метод, 81, 82, 141, 203, 218, 330, 336, 338
 - Buffer, тип, 122
 - String() (тип Regexp), 168
 - String() (тип Value), 538
 - Stringer, интерфейс (пакет fmt), 336, 338, 361
 - Strings()
 - пакет sort, 312
 - strings, пакет, 125, 145, 527
 - Contains(), 147
 - Count(), 147, 225
 - EqualFold(), 147, 213
 - Fields(), 63, 109, 147, 150
 - FieldsFunc(), 146, 147, 231
 - HasPrefix(), 147
 - HasPrefix(), функция, 330
 - HasSuffix(), 147, 176
 - Index(), 147, 179
 - IndexAny(), 147, 179
 - IndexFunc(), 127, 128, 147
 - IndexRune(), 147
 - Join(), 31, 82, 147, 150
 - LastIndex(), 147
 - LastIndexAny(), 147
 - LastIndexFunc(), 127, 147
 - Map(), 147, 151, 179, 195
 - NewReader(), 147, 152
 - NewReplacer(), 147
 - Repeat(), 136
 - Replace(), 109, 148, 149
 - Split(), 62, 63, 146, 148
 - SplitAfter(), 146, 148
 - SplitAfterN(), 146, 148
 - SplitN(), 146, 148
 - Title(), 148
 - ToLower(), 148, 211
 - ToLowerSpecial(), 148
 - WithTitle(), 148
 - WithTitleSpecial(), 148
 - ToUpper(), 148
 - ToUpperSpecial(), 148
 - Trim(), 148
 - TrimFunc(), 148
 - TrimLeft(), 148
 - TrimLeftFunc(), 149
 - TrimRight(), 149, 422
 - TrimRightFunc(), 149
 - TrimSpace(), 149
 - Strings() (пакет sort), 210
 - StringsAreSorted() (пакет sort), 210
 - struct, ключевое слово, 67, 176, 349
 - SubexpNames() (тип Regexp), 168
 - switch, инструкция, 146, 246, 247, 249, 257, 261, 300, 356, 361, 531
 - sync, пакет
 - Mutex, тип, 430
 - RWMutex, тип, 430
 - WaitGroup, тип, 441
- ## T
- Tan() (пакет cmplx), 102
 - Tan() (пакет math), 96
 - Tanh() (пакет cmplx), 102
 - Tanh() (пакет math), 96
 - tag, пакет (пакет archive)
 - Header, тип, 504, 507
 - Reader, тип, 507
 - Writer, тип, 503
 - TempFile() (пакет ioutil), 532
 - template, пакет (пакет html), 528
 - template, пакет (пакет text), 528

- testing, пакет, 519
- Tick() (пакет time), 536
- time, пакет, 536
 - After(), 420, 536
 - Duration, тип, 420
 - NewTicker(), 536
 - Parse(), 474, 479, 482
 - Tick(), 536
 - Time, тип, 457, 497
 - Unix(), 492
- Time, тип()
 - Format(), 492
- Time, тип (пакет time), 457, 497
 - Format(), 464, 478
 - Unix(), 492
- Title() (пакет strings), 148
- ToLowerSpecial() (пакет strings), 148
- ToLower() (пакет strings), 148, 211
- ToLower() (пакет unicode), 160
- ToTitleSpecial() (пакет strings), 148
- ToTitle() (пакет strings), 148
- ToTitle() (пакет unicode), 160
- ToUpperSpecial() (пакет strings), 148
- ToUpper() (пакет strings), 148
- ToUpper() (пакет unicode), 160
- To() (пакет unicode), 160
- TrimFunc() (пакет strings), 148
- TrimLeftFunc() (пакет strings), 149
- TrimLeft() (пакет strings), 148
- TrimRightFunc() (пакет strings), 149
- TrimRight() (пакет bytes), 422
- TrimRight() (пакет strings), 149, 422
- TrimSpace() (пакет strings), 149
- Trim() (пакет strings), 148
- Trunc() (пакет math), 96
- TypeOf() (пакет reflect), 537
- type
 - инструкция, 518
 - ключевое слово, 255, 326
- U**
- uint (тип встроенный), 87, 99
- uint8 (тип встроенный), 87, 142
- uint16 (тип встроенный), 87
- uint32 (тип встроенный), 87
- uint64 (тип встроенный), 87
- uintptr (тип встроенный), 87
- unicode, пакет, 145, 160, 527
 - Is(), 160
 - IsControl(), 160
 - IsDigit(), 160
 - IsGraphic(), 160
 - IsLetter(), 160
 - IsLower(), 160
 - IsMark(), 160
 - IsOneOf(), 160
 - IsPrint(), 160
 - IsPunct(), 160
 - IsSpace(), 128, 150, 160
 - IsSymbol(), 160
 - IsTitle(), 160
 - IsUpper(), 160
 - MaxRune, константа, 113
 - SimpleFold(), 160
 - To(), 160
 - ToLower(), 160
 - ToTitle(), 160
 - ToUpper(), 160
- UniqueInts(), упражнение, 235
- Unix()
 - пакет time, 492
 - тип Time, 492
- Unlock() (тип RWMutex), 437
- Unmarshaler, интерфейс (пакет json), 466
- Unmarshal() (пакет json), 255, 466
- unpack, пример, 499
- unpack, упражнение, 509
- Unquote() (пакет strconv), 156, 158
- UnquoteChar() (пакет strconv), 156
- url, пакет (пакет net), 537
- US-ASCII, кодировка, 114
- utf8, пакет
 - DecodeLastRuneInString(), 295
 - DecodeRuneInString(), 295
- utf8, пакет (пакет unicode), 145, 158, 527

DecodeLastRune(), 159
DecodeLastRuneInString(), 126, 159
DecodeRune(), 159
DecodeRuneInString(), 126, 159
EncodeRune(), 159
FullRune(), 159
FullRuneInString(), 159
RuneCount(), 159
RuneCountInString(), 136, 159
RuneLen(), 159
RuneStart(), 159
Valid(), 159
ValidString(), 159
utf16-to-utf8/utf16-to-utf8, упражнение, 510
utf16, пакет (пакет unicode), 527

V

Valid() (пакет utf8), 159
ValidString() (пакет utf8), 159
Value, тип (пакет reflect)
 Bool(), 538
 Call(), 539
 Complex(), 538
 Elem(), 539
 Float(), 538
 Int(), 538
 Len(), 541
 MethodByName(), 541
 String(), 538
ValueOf() (пакет reflect), 537
var, ключевое слово, 34, 78, 241
Version() (пакет runtime), 533
Vorbis Audio, аудиоформат, 174

W

Wait() (тип WaitGroup), 400, 447
WaitGroup, тип (пакет sync), 441
 Add(), 443, 445, 447
 Done(), 443, 445, 447
 Wait(), 400, 447

Walk() (пакет filepath), 442, 445
wordfrequency, пример, 227
WriteAt() (тип File), 499
WriteCloser, интерфейс (пакет io), 503, 508
WriteFile() (пакет ioutil), 533
Writer, интерфейс (пакет io), 54, 56, 129, 324, 447, 461, 462, 470, 485, 490, 508
Writer, тип (пакет tar), 503
Writer, тип (пакет zip), 499
Write() (пакет binary), 489, 495
WriteString(), тип Buffer, 122, 257

X

xml, пакет
 Header, константа, 470
 Name, тип, 471
xml, пакет (пакет encoding), 467, 468
 Name, тип, 468
XML, формат, 458, 467

Y

Y0() (пакет math), 96
Y1() (пакет math), 96
Yn() (пакет math), 96

Z

zip, пакет (пакет archive), 499
 FileInfo(), 506
 FileInfoHeader(), 501
 File, тип, 505
 NewReader(), 505
 ReadCloser, тип, 505
 Writer, тип, 499
ZP, нулевая точка (пакет image), 367

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС БУКС» наложенным платежом, выслав открытку или письмо по почтовому адресу: 123242, Москва, а/я 20 или по электронному адресу: **orders@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество покупателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. (499) 725-54-09, 725-50-27; электронный адрес **books@alians-kniga.ru**.

Марк Саммерфильд

**Программирование на Go
Разработка приложений XXI века**

Главный редактор *Мовчан Д. А.*
dm@dmk-press.ru

Корректор *Синяева Г. И.*

Верстка *Татаринов А. Ю.*

Дизайн обложки *Мовчан А. Г.*

Подписано в печать 28.01.2013. Формат 60×90 1/16.

Гарнитура «Петербург». Печать офсетная.

Усл. печ. л. 36,5. Тираж 200 экз.

Веб-сайт издательства: www.dmk-press.ru

Практическое руководство по Go – революционно новому языку программирования со встроенной поддержкой параллельного программирования, упрощающему разработку программ для многопроцессорных систем.

На сегодняшний день Go – самый восхитительный из новых языков программирования. В этом руководстве первопроходец языка Go Марк Саммерфильд показывает, как писать программы, в полной мере использующие его революционные возможности и идиомы.

Будучи одновременно и учебником, и справочником, эта книга соединяет в себе все знания, которые потребуются, чтобы продолжить освоение Go, думать на Go и писать высокопроизводительные программы на Go. Саммерфильд приводит множество сравнений идиом программирования, демонстрируя преимущества Go, уделяя особое внимание ключевым инновациям. Начиная с самых основ, он разъясняет все аспекты параллельного программирования на языке Go с применением каналов и без использования блокировок, а также гибкость и необычность подхода к объектно-ориентированному программированию с применением механизма динамической типизации.

Основной упор автор делает на практическое применение языка. Каждая глава содержит выверенные, действующие примеры программного кода, чтобы помочь читателю быстрее овладеть мастерством разработки. Везде, где это возможно, приводятся законченные программы и пакеты, демонстрирующие особенности практического применения, а также упражнения.

В этой книге:

- рассказывается, как получить и установить Go, и как собирать и запускать программы;
- исследуются синтаксис языка Go, его особенности и обширная стандартная библиотека;
- описываются логические значения, выражения и числовые типы;
- демонстрируются приемы создания, сравнения, индексирования и форматирования строк;
- дается полное представление об очень эффективных встроенных типах коллекций, таких как срезы и отображения;
- демонстрируются приемы процедурного программирования на языке Go;
- раскрываются необычные и гибкие особенности подхода к объектно-ориентированному программированию;
- рассказывается об уникальном, простом и естественном подходе к параллельному программированию;
- описываются приемы чтения и записи двоичных и текстовых файлов, а также файлов в форматах JSON и XML;
- демонстрируются возможности импортирования и использования пакетов из стандартной библиотеки, а также пользовательских и сторонних пакетов;
- раскрываются возможности создания документации, модульного тестирования и оценки производительности пользовательских пакетов.

Марк Саммерфильд (Mark Summerfield) – владелец компании Qtrac Ltd., является независимым инструктором, консультантом, техническим редактором и писателем, специализирующимся на языках программирования Go, Python, C++, а также на библиотеках Qt и PyQt. Его перу принадлежат такие книги, как «Rapid GUI Programming with Python and Qt» (Prentice Hall, 2007), «C++ GUI Programming with Qt 4» (Prentice Hall, 2008) – «Qt 4: Программирование GUI на C++. 2-е издание» (Кудиц-Пресс, 2008). «Programming in Python 3, Second Edition» (Addison-Wesley, 2009) – «Программирование на Python 3. Подробное руководство» (Символ-Плюс, 2009) и «Advanced Qt Programming» (Prentice Hall, 2010) – «Qt. Профессиональное программирование. Разработка кроссплатформенных приложений на C++» (Символ-Плюс, 2011).

Internet-магазин: www.dmk-press.ru
Книга - почтой: orders@aliants-kniga.ru
Оптовая продажа: «Альянс-книга»
тел. (499)725-5409
books@aliants-kniga.ru

ДМК
ИЗДАТЕЛЬСТВО
www.dmk-press.ru

ISBN 978-5-94074-854-0



9 785940 748540 >