William Kennedy

WITH Brian Ketelsen Erik St. Martin FOREWORD BY Steve Francia

MANNING

www.allitebooks.com

*Go in Action*

Licensed to Mark www.allitebooks.com

Watson <nordickan@gmail.com>

Licensed to Mark www.allitebooks.com

Watson <nordickan@gmail.com>

*Go in Action*

WILLIAM KENNEDY

WITH BRIAN KETELSEN AND ERIK ST. MARTIN

MANNING SHELTER ISLAND

Licensed to Mark www.allitebooks.com

Watson <nordickan@gmail.com>

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department Manning Publications Co. 20 Baldwin Road PO Box 761 Shelter Island, NY 11964 Email: orders@manning.com

©2016 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

Recognizing the importance of preserving what has been written, it is Manning’s policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

Manning Publications Co. Development editor: Jennifer Stout 20 Baldwin Road Technical development editor: Kim Shrier PO Box 761 Copyeditor: Jodie Allen Shelter Island, NY 11964 Proofreader: Katie Tennant

Technical proofreader: Jimmy Frasché Typesetter: Dottie Marsico Cover designer: Marija Tudor

ISBN 9781617291784 Printed in the United States of America 1 2 3 4 5 6 7 8 9 10 – EBM – 20 19 18 17 16 15

Licensed to Mark Watson <nordickan@gmail.com>

www.allitebooks.com

*brief contents*

*1* ■ Introducing Go 1 *2* ■ Go quick-start 9 *3* ■ Packaging and tooling 39 *4* ■ Arrays, slices, and maps 57 *5* ■ Go’s type system 88 *6* ■ Concurrency 128 *7* ■ Concurrency patterns 158 *8* ■ Standard library 184 *9* ■ Testing and benchmarking 211

Licensed to Mark Watson <nordickan@gmail.com>

www.allitebooks.com

**v**

Licensed to Mark www.allitebooks.com

Watson <nordickan@gmail.com>

*contents*

*foreword xi preface xiii acknowledgments xiv about this book xvi about the cover illustration xix 1* ***Introducing Go 1***

1.1 Solving modern programming challenges with Go 2

*Development speed 3* ■ *Concurrency 3* ■ *Go’s type system 5 Memory management 7* 1.2 Hello, Go 7

*Introducing the Go Playground 8* 1.3 Summary 8 *2* ***Go quick-start 9***

2.1 Program architecture 10 2.2 Main package 11 2.3 Search package 13

*search.go 14* ■ *feed.go 22* ■ *match.go/default.go 26* 2.4 RSS matcher 32 2.5 Summary 38

Licensed to Mark Watson <nordickan@gmail.com>

www.allitebooks.com

**vii**

CONTENTS **viii**

*3* ***Packaging and tooling 39*** 3.1 Packages 40

*Package-naming conventions 40* ■ *Package main 40* 3.2 Imports 42

*Remote imports 42* ■ *Named imports 43* 3.3 init 44 3.4 Using Go tools 45 3.5 Going farther with Go developer tools 47

*go vet 47* ■ *Go format 48* ■ *Go documentation 48* 3.6 Collaborating with other Go developers 51

*Creating repositories for sharing 51* 3.7 Dependency management 52

*Vendoring dependencies 52* ■ *Introducing gb 54* 3.8 Summary 56 *4* ***Arrays, slices, and maps 57***

4.1 Array internals and fundamentals 57

*Internals 58* ■ *Declaring and initializing 58* ■ *Working with arrays 60* ■ *Multidimensional arrays 62* ■ *Passing arrays between functions 64*

4.2 Slice internals and fundamentals 65

*Internals 65* ■ *Creating and initializing 65* ■ *Working with slices 68* ■ *Multidimensional slices 79* ■ *Passing slices between functions 80*

4.3 Map internals and fundamentals 81

*Internals 81* ■ *Creating and initializing 83* ■ *Working with maps 84* ■ *Passing maps between functions 86* 4.4 Summary 87 *5* ***Go’s type system 88***

5.1 User-defined types 89 5.2 Methods 92 5.3 The nature of types 96

*Built-in types 96* ■ *Reference types 97* ■ *Struct types 98*

Licensed to Mark Watson <nordickan@gmail.com>

www.allitebooks.com

CONTENTS **ix**

5.4 Interfaces 101

*Standard library 102* ■ *Implementation 104* ■ *Method sets 105* ■ *Polymorphism 109* 5.5 Type embedding 111 5.6 Exporting and unexporting identifiers 119 5.7 Summary 127 *6* ***Concurrency 128***

6.1 Concurrency versus parallelism 129 6.2 Goroutines 132 6.3 Race conditions 139 6.4 Locking shared resources 142

*Atomic functions 142* ■ *Mutexes 145* 6.5 Channels 147

*Unbuffered channels 148* ■ *Buffered channels 153* 6.6 Summary 157 *7* ***Concurrency patterns 158*** 7.1 Runner 158 7.2 Pooling 167 7.3 Work 177 7.4 Summary 183 *8* ***Standard library 184***

8.1 Documentation and source code 185 8.2 Logging 187

*Log package 187* ■ *Customized loggers 191 Conclusion 195* 8.3 Encoding/Decoding 196

*Decoding JSON 196* ■ *Encoding JSON 201 Conclusion 202* 8.4 Input and output 203

*Writer and Reader interfaces 203* ■ *Working together 205 Simple curl 208* ■ *Conclusion 210* 8.5 Summary 210

Licensed to Mark Watson <nordickan@gmail.com>

www.allitebooks.com

CONTENTS **x**

*9* ***Testing and benchmarking 211***

9.1 Unit testing 212

*Basic unit test 212* ■ *Table tests 216* ■ *Mocking calls 219 Testing endpoints 223* 9.2 Examples 228 9.3 Benchmarking 232 9.4 Summary 236

*index 237*

Licensed to Mark Watson <nordickan@gmail.com>

*foreword*

In computer science, when you think of exceptional people, a few names come to mind. Among them are Rob Pike, Robert Griesmier, and Ken Thompson, who are responsible for UNIX, Plan 9, B, Java’s JVM HotSpot, V8, Strongtalk, Sawzall, Ed, Acme, and UTF8, among many other creations. In 2007, they came together to experi- ment with a very powerful idea, combining their decades of experience to create a new systems language inspired by existing languages but truly unlike anything that came before. They released their creation as open source and named it “Go.” If Go continues on the course it is now on, it may indeed prove to be the most impactful of their many notable creations.

Humanity is at its best when people join together with the pure intention of mak- ing the world a better place. In 2013, Brian and Erik formed the Gopher Academy and were soon joined by Bill and a few other similar-minded people, united in the pursuit of building a better community around the Go language. They first noticed that the community needed a place to gather and share material online so they set up the Go discussion board (slack) and the Gopher Academy blog. As time went on and the community continued to grow, they established the world’s first global Go confer- ence, GopherCon. Through their deep experience with the community, they knew that a resource was needed to guide the many thousands of programmers into this new language, so they began to write the book that you now hold in your hands.

This book is a labor of love from three individuals who have given so much of their time and talents to the Go community. I have been alongside Bill, Brian, and Erik to witness them writing and revising over the past year as they maintained their existing responsibilities as editors of the Gopher Academy blog, as conference organizers, in their day jobs, and in their roles as fathers and husbands. To them this is not a book,

Licensed to Mark Watson <nordickan@gmail.com>

**xi**

FOREWORD **xii**

but a tribute to the language they love. They weren’t content with producing a “good” book. They wrote and reviewed, rewrote and revised many drafts of each page, exam- ple, and chapter until they had a book worthy of the language they hold so dear.

It takes courage to leave a language of comfort and familiarity and try a language that is not only new to you but new to the world. This road less traveled is a bumpy one, lined with bugs that only early adopters are familiar with. It includes unexpected errors, spotty or missing documentation, and a lack of established libraries to use. This is the path of a trailblazer, a pioneer. If you are reading this now, you are likely on the beginning of this journey.

From the first chapter to the last, this book is crafted to provide you, the reader, a concise and comprehensive guide to exploring, learning, and using Go. In all the world, you couldn’t hope to have better guides than Bill, Brian, and Erik. I’m excited for you to discover all the goodness that is Go and look forward to seeing you online and at the Go meetups and conferences.

STEVE FRANCIA GOPHER AND CREATOR OF HUGO, COBRA, VIPER, AND SPF13-VIM

Licensed to Mark Watson <nordickan@gmail.com>

*preface*

Back in October 2013 after writing the GoingGo.net blog for a few months, I received a call from Brian Ketelsen and Erik St. Martin. They were in the process of writing this book and asked if I would be a part of it. I jumped at the opportunity and started writ- ing. I was still very new to Go at the time, so this was a great chance to learn more about the language, work with Brian and Erik and share what I learned at a greater scale than the blog.

After we finished the first four chapters, we released the book under the Manning Early Access Program (MEAP). Soon after, we received an email from a member of the language team. This person provided a review that contained a detailed set of changes plus a wealth of knowledge, advice, encouragement, and support. From there, we decided to rewrite chapter 2 from scratch and performed a major overhaul of chapter 4. We learned that rewriting chapters was not going to be the exception but the norm. That experience also taught us that it was going to take the help of the community to write this book, and we needed to make that happen immediately.

Ever since then, this book has been a community effort. We have tried to put a proper amount of time in researching each chapter, developing code samples, and working with the community to review, discuss, and edit the material and code. We have done our best to make sure this book is technically correct, shows only idiomatic code, and teaches you Go the way the community feels it should be written and thought about. We do have some of our own thoughts, practices, and guidelines sprinkled in as well.

We hope this book helps you learn Go and you find it a useful resource today and for many years to come. Brian, Erik, and I are always online and available to help any- one who reaches out to us. If you purchased the book, thank you, and don’t be shy about saying “hi.”

WILLIAM KENNEDY

Licensed to Mark Watson <nordickan@gmail.com>

**xiii**

**xiv***acknowledgments* We have spent over 18 months writing this book, but none of our efforts would have been possible without the support of many people—our families, friends, colleagues, and mentors; the entire Go community; and our publisher, Manning.

When you’re writing a book like this, you need an editor who will not only share the good but help you through the bad and be there for you at all cost. Jennifer Stout, you’re a brilliant, nurturing, and amazing friend. Thank you for everything and for being there when we needed you the most. Thank you for making this book a reality. Thanks also to all the other folks at Manning who worked with us during the develop- ment and production of our book.

You can’t know everything, so it requires a community of people to give their time and knowledge. We thank the Go community and everyone who participated in reviews and provided feedback on the manuscript at various stages of its development, especially Adam McKay, Alex Basile, Alex Jacinto, Alex Vidal, Anjan Bacchu, Benoît Benedetti, Bill Katz, Brian Hetro, Colin Kennedy, Doug Sparling, Jeffrey Lim, Jesse Evans, Kevin Jackson, Mark Fisher, Matt Zulak, Paulo Pires, Peter Krey, Philipp K. Janert, Sam Zaydel, and Thomas O’Rourke. Thanks also to Jimmy Frasché for his care- ful technical review of the final manuscript shortly before it went into production.

There are a few other people who need to be acknowledged in particular. Kim Shrier was there from the very beginning, providing reviews, and giving of his time to teach. We learned so many things from you and we are grateful. The book is better technically because of you.

Bill Hathaway got involved heavily in the last year of writing the book, shaping each chapter; his thoughts and opinions were invaluable. We must give Bill credit as a coauthor of chapter 9. It would not exist without Bill’s time, talent, and effort.

Licensed to Mark Watson <nordickan@gmail.com>

ACKNOWLEDGMENTS **xv**

We would also like to recognize Cory Jacobson, Jeffery Lim, Chetan Conikee, and Nan Xiao, who consistently provided time for reviews, opinions, and guidance. Thanks to Gabriel Aszalos, Fatih Arslan, Kevin Gillette, and Jason Waldrip for help with sample code and reviews. And special thanks to Steve Francia for contributing the foreword and endorsing our work.

We end by sincerely thanking our families and friends. Anything that takes this level of commitment and time always has an effect on the ones you love.

WILLIAM KENNEDY I would like to thank Lisa, my beautiful wife, and my five children: Brianna, Melissa, Amanda, Jarrod, and Thomas. Lisa, I know you and the kids spent way too many days, nights, and weekends without your husband and father. Thank you for letting me take all the time I needed to work on the book: I love each and every one of you.

I would also like to thank my business partner Ed Gonzalez, creative director Erick Zelaya, and the entire team at Ardan Studios. Ed, thanks for supporting me from the beginning. I could not have done this without you. You are more than just a business partner, you are a friend and brother: thank you. Erick, thanks for everything you do to support me and the company. Not sure what we would do without you.

BRIAN KETELSEN I would like to thank my family for their patience during this four-year-long process of producing a book. Christine, Nathan, Lauren, and Evelyn: thank you for putting up with me as I wrote chapters in a lounge chair by the pool while you were swimming.

Thank you for believing that this book could and would be published.

ERIK ST. MARTIN I would like to thank my fiancée Abby, and my three children Halie, Wyatt, and Allie for being so patient and understanding how much time writing a book and organizing conferences demand. I love you all so very much and am lucky to have you.

I would also like to thank Bill Kennedy for the tremendous effort he has poured into this book—we asked him to help us write it, and he steered the ship most of the way due to the demands of our jobs and organizing GopherCon. I also want to thank the community for all their reviews and words of encouragement.

Licensed to Mark Watson <nordickan@gmail.com>

*about this book*

Go is an open source programming language that makes it easy to build simple, reli- able, and efficient software. Although it borrows ideas from existing languages, it has a unique and simple nature that makes Go programs different in character from pro- grams written in other languages. It balances the capabilities of a low-level systems lan- guage with some high-level features you see in modern languages today. This creates a programming environment that allows you to be incredibly productive, performant, and fully in control; in Go, you can write less code and do so much more.

***Who should read this book?*** This book was written for an intermediate-level developer who has some experience with other programming languages and wants to learn Go. Our goal in writing this book is to provide you an intensive, comprehensive, and idiomatic view of the lan- guage. We focus on both the specification and implementation of the language, including topics that range from language syntax, Go’s type system, concurrency, channels, testing, and more. We believe this book is perfect for anyone who wants a jump-start in learning Go as well as for those who want a more thorough understand- ing of the language and its internals.

***Roadmap***

The book consists of nine chapters, briefly described here:

▪ Chapter 1 is a quick introduction to what the language is, why it was created, and the problems it solves. It also briefly introduces some of Go’s core concepts such as concurrency.

Licensed to Mark Watson <nordickan@gmail.com>

**xvi**

ABOUT THIS BOOK **xvii**

▪ Chapter 2 walks you through a complete Go program, teaching you all that Go has to offer as a language along the way.

▪ Chapter 3 introduces the concept of packaging and how to best set up your Go workspace and development environment. It also shows how to use the Go tool- ing, including fetching and building your code.

▪ Chapter 4 provides a detailed view of Go’s built-in data types: arrays, slices, and maps. It explains the implementation and mechanics behind these data structures.

▪ Chapter 5 is a detailed view of Go’s type system, from struct types to named types to interfaces and type embedding. It also covers how all these things come together to allow you to structure and write complex software in a simpler way.

▪ Chapter 6 dives deeply into how the Go scheduler, concurrency, and channels work. It teaches the mechanics behind this aspect of the language.

▪ Chapter 7 takes what you learn from chapter 6 and shows more practical code around concurrency patterns. You will learn how to implement goroutine pools to manage work and how to pool reusable resources to be shared.

▪ Chapter 8 explores the standard library and goes deep into three packages: log, json, and io. The chapter focuses on some of the intricacies of these three packages.

▪ Chapter 9 closes the book by showing how to use the testing and benchmarking framework. You will learn how to write unit and table tests and benchmarks, and how to add examples to your documentation and use the examples as tests.

***About the code***

All source code in the book is presented in a mono-spaced typeface like this, which sets it off from the surrounding text. In many listings, the code is annotated to point out key concepts, and numbered bullets are sometimes used in the text to provide additional information about the code.

Source code for the examples in the book is available for download from the pub- lisher’s website at www.manning.com/books/go-in-action and from GitHub at https:// github.com/goinaction/code.

***Author Online*** Purchase of *Go in Action* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the authors and from other users. To access the forum and sub- scribe to it, point your web browser to www.manning.com/books/go-in-action. This page provides information on how to get on the forum once you’re registered, what kind of help is available, and the rules of conduct on the forum.

Licensed to Mark Watson <nordickan@gmail.com>

ABOUT THIS BOOK **xviii**

Manning’s commitment to our readers is to provide a venue where a meaningful dialog between individual readers and between readers and the authors can take place. It is not a commitment to any specific amount of participation on the part of the authors, whose contributions to the AO remain voluntary (and unpaid). We sug- gest you ask the authors challenging questions, lest their interest stray.

The Author Online forum and the archives of previous discussions will be accessi- ble from the publisher’s website as long as the book is in print.

***About the authors***

WILLIAM KENNEDY (@goinggodotnet) is a managing partner at Ardan Studio in Miami, Florida, a mobile, web, and systems development company. He is also the author of the blog GoingGo.Net, and the organizer for the Go meetup in Miami. Bill is focused on Go education through his training company, Ardan Labs. He can often be found talking at conferences and giving workshops both locally and over hangouts. He always finds time to work with individuals and groups who want to take their Go knowledge, blogging, and coding skills to the next level.

BRIAN KETELSEN (@bketelsen) is the CIO and cofounder of XOR Data Exchange. Brian is a co-organizer of GopherCon, the annual Go conference, and the founder of GopherAcademy—a community-focused organization created for the promotion of the Go language and the education of Go developers. He’s been using Go since 2010.

ERIK ST. MARTIN (@erikstmartin) is the Director of Software Development at XOR Data Exchange, a big data and analytics company located in Austin, Texas, but resides in Tampa, Florida. Erik is a long-time contributor to open source and its communities. He’s an organizer for GopherCon, an annual Go conference, and the organizer of the Go Tampa meetup group. He’s very passionate about Go and the community and eager to find new ways to foster its growth.

Licensed to Mark Watson <nordickan@gmail.com>

*about the cover illustration*

The figure on the cover of *Go in Action* is captioned “Man from the East Indies.” The illustration is taken from Thomas Jefferys’ *A Collection of the Dresses of Different Nations, Ancient and Modern* (four volumes), London, published between 1757 and 1772. The title page states that these are hand-colored copperplate engravings, heightened with gum arabic. Thomas Jefferys (1719–1771) was called “Geographer to King George III.” He was an English cartographer who was the leading map supplier of his day. He engraved and printed maps for government and other official bodies and produced a wide range of commercial maps and atlases, especially of North America. His work as a map maker sparked an interest in local dress customs of the lands he surveyed and mapped, and which are brilliantly displayed in this collection.

Fascination with faraway lands and travel for pleasure were relatively new phenom- ena in the late eighteenth century, and collections such as this one were popular, introducing both the tourist as well as the armchair traveler to the inhabitants of other countries. The diversity of the drawings in Jefferys’ volumes speaks vividly of the uniqueness and individuality of the world’s nations some 200 years ago. Dress codes have changed since then, and the diversity by region and country, so rich at the time, has faded away. It is now often hard to tell the inhabitants of one continent from another. Perhaps, trying to view it optimistically, we have traded a cultural and visual diversity for a more varied personal life—or a more varied and interesting intellectual and technical life.

At a time when it is hard to tell one computer book from another, Manning cele- brates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by Jeffreys’ pictures.

Licensed to Mark Watson <nordickan@gmail.com>

www.allitebooks.com

**xix**

Licensed to Mark Watson <nordickan@gmail.com>

*Introducing Go*

***In this chapter***

▪ Solving modern computing challenges with Go

▪ Using the Go tools

Computers have evolved, but programming languages haven’t kept up the same pace of evolution. The cell phones we carry might have more CPU cores than the first computer we used. High-powered servers now have 64, 128, or even more cores, but we’re still programming using the techniques we were using for a single core.

The art of programming has evolved too. Most programs aren’t written by a sin- gle developer any more: they’re written by teams of people sitting in different time zones and working at different times of the day. Large projects are broken up into smaller pieces and assigned to programmers who then deliver their work back to the team in the form of a library or package that can be used across an entire suite of applications.

Today’s programmers and companies believe more than ever in the power of open source software. Go is a programming language that makes sharing code easy. Go ships with tools that make it simple to use packages written by others, and Go makes it easy to share our own packages too.

Licensed to Mark Watson <nordickan@gmail.com>

**1**

**2** CHAPTER 1 ***Introducing Go***

In this chapter you’ll see how Go is different from other programming languages. Go rethinks the traditional object-oriented development you might be used to, while still providing an efficient means for code reuse. Go makes it easier for you to effec- tively use all of the cores on your expensive server, and it takes away the penalty of compiling a very large project.

As you read this chapter, you’ll get a feeling for the many decisions that shaped the creation of Go, from its concurrency model to its lightning-fast compiler. We mentioned it in the preface, but it bears repeating: this book has been written for an intermediate-level developer who has some experience with other programming lan- guages and wants to learn Go. Our goal in writing this book is to provide you an inten- sive, comprehensive, and idiomatic view of the language. We focus on both the specification and implementation of the language, including the wide-ranging topics of language syntax, Go’s type system, concurrency, channels, testing, and more. We believe this book is perfect for anyone who wants a jump-start in learning Go or who wants a more thorough understanding of the language and its internals.

The source code for the examples in the book is available at https://github.com/ goinaction/code.

We hope you’ll appreciate the tools that ship with Go to make your life as a devel- oper easier. In the end, you’ll appreciate why so many developers are choosing Go when they start up that new project.

***1.1 Solving modern programming challenges with Go***

The Go team went to great lengths to solve the problems facing software developers today. Developers have to make an uncomfortable choice between rapid development and performance when choosing a language for their projects. Languages like C and C++ offer fast execution, whereas languages like Ruby and Python offer rapid develop- ment. Go bridges these competing worlds and offers a high-performance language with features that make development fast.

As we explore Go, you’ll find well-planned features and concise syntax. As a lan- guage, Go is defined not only by what it includes, but by what it doesn’t include. Go has a concise syntax with few keywords to memorize. Go has a compiler that’s so fast, sometimes you’ll forget it’s running. As a Go developer, you’ll spend significantly less time waiting for your project to build. Because of Go’s built-in concurrency features, your software will scale to use the resources available without forcing you to use spe- cial threading libraries. Go uses a simple and effective type system that takes much of the overhead out of object-oriented development and lets you focus on code reuse. Go also has a garbage collector, so you don’t have to manage your own memory. Let’s look quickly at these key features.

Licensed to Mark Watson <nordickan@gmail.com>

**3 *Solving modern programming challenges with Go***

***1.1.1 Development speed***

Compiling a large application in C or C++ takes more time than getting a cup of coffee. Figure 1.1 shows an XKCD classic excuse for messing around in the office. Go offers lightning-quick compiles by using a smart compiler and simplified dependency resolu- tion algorithms. When you build a Go program, the compiler only needs to look at the libraries that you directly include, rather than traversing the depen- dencies of all the libraries that are included in the entire dependency chain like Java, C, and C++. Con- sequently, many Go applications compile in under a second. The entire Go source tree compiles in under 20 seconds on modern hardware.

Figure 1.1 Working hard? (via XKCD)

Writing applications in dynamic languages makes you productive quickly because there are no intermediate steps between writing code and executing it. The trade-off is that dynamic languages don’t offer the type safety that static languages do and often need a comprehensive test suite to avoid discovering incorrect type bugs at runtime.

Imagine writing a large application in a dynamic language like JavaScript and com- ing across a function that expects to receive a field called ID. Is that an integer, a string, or a UUID? The way to find out is to look at the source. You could try to execute the function with a number or a string and see what happens. In Go, you wouldn’t spend time wondering, because the compiler will catch type differences for you.

***1.1.2 Concurrency***

One of the hardest things to do as a programmer is to write an application that effec- tively uses the available resources of the hardware running it. Modern computers have many cores, but most programming languages don’t have effective tools for utilizing those additional resources easily. They often require a lot of thread synchronization code, which is prone to errors.

Go’s concurrency support is one of its strongest features. Goroutines are like threads, but use far less memory and require less code to use. Channels are data struc- tures that let you send typed messages between goroutines with synchronization built in. This facilitates a programming model where you send data between goroutines, rather than letting the goroutines fight to use the same data. Let’s look at these fea- tures in more detail now.

GOROUTINES Goroutines are functions that run concurrently with other goroutines, including the entry point of your program. In other languages, you’d use threads to accomplish the same thing, but in Go many goroutines execute on a single thread. For example, if you write a web server and you want to handle different web requests simultaneously, you’d have to write a lot of extra code to use threads in C or Java. In Go, the net/http library

Licensed to Mark Watson <nordickan@gmail.com>

**4** CHAPTER 1 ***Introducing Go***

Goroutine Goroutine Goroutine

Goroutine Goroutine Thread

has concurrency built in using goroutines. Each inbound request automatically runs on its own goroutine. Goroutines use less memory than threads and the Go runtime will automatically schedule the execution of goroutines against a set of configured logical processors. Each logical processor is bound to a single OS thread (see figure 1.2). This makes your application much more efficient with significantly less development effort. If you want to execute some code concurrently while you move on to accomplish other things, a goroutine is perfect for the job. Here’s a quick example:

func log(msg string){

... some logging code here }// Elsewhere in our code after we've discovered an error. go log("something dire happened")

That keyword go is all you need to schedule the log function to run as a goroutine and for that goroutine be run concurrently with other goroutines. This means you can continue executing the rest of your application while the logging happens concur- rently, which often results in greater perceived performance for your end users. As stated before, goroutines have minimal overhead, so it isn’t uncommon to spawn tens of thousands of them. We’ll explore goroutines and concurrency more in-depth in chapter 6.

CHANNELS Channels are data structures that enable safe data communication between gorou- tines. Channels help you to avoid problems typically seen in programming languages that allow shared memory access.

The hardest part of concurrency is ensuring that your data isn’t unexpectedly modified by concurrently running processes, threads, or goroutines. When multiple threads change the same data without locks or synchronization, heartache always fol- lows. In other languages, when you have global variables and shared memory, you’re required to use complicated locking disciplines to prevent unsynchronized changes to the same variables.

Goroutine

Goroutine

Goroutine

Goroutine

Thread

Thread

Figure 1.2 Many goroutines execute on a single OS thread

Licensed to Mark Watson <nordickan@gmail.com>

**5 *Solving modern programming challenges with Go***

Goroutine

Goroutine

Goroutine

Channel

Channel

Data

Data

Figure 1.3 Using channels to safely pass data between goroutines

Channels help to solve this problem by providing a pattern that makes data safe from concurrent modification. Channels help to enforce the pattern that only one gorou- tine should modify the data at any time. You can see an example of this flow in figure 1.3, where channels are used to send data between several running goroutines. Imagine an application where many different processes need to know about or modify data sequentially. Using goroutines and channels, you can model this process safely.

In figure 1.3 you see three goroutines and two unbuffered channels. The first gor- outine passes a data value through the channel to a second goroutine that’s already waiting. The exchange of the data between both goroutines is synchronized, and once the hand-off occurs, both goroutines know the exchange took place. After the second goroutine performs its tasks with the data, it then sends the data to a third goroutine that’s waiting. That exchange is also synchronized, and both goroutines can have guarantees the exchange has been made. This safe exchange of data between gorou- tines requires no other locks or synchronization mechanisms.

It’s important to note that channels don’t provide data access protection between goroutines. If copies of data are exchanged through a channel, then each goroutine has its own copy and can make any changes to that data safely. When pointers to the data are being exchanged, each goroutine still needs to be synchronized if reads and writes will be performed by the different goroutines.

***1.1.3 Go’s type system***

Go provides a flexible hierarchy-free type system that enables code reuse with minimal refactoring overhead. It’s still object-oriented development, but without the tradi- tional headaches. If you’ve ever spent a week planning your abstract classes and inter- faces in a complex Java or C++ program, you’ll appreciate the simplicity of Go’s type system. Go developers simply embed types to reuse functionality in a design pattern called *composition*. Other languages use composition, but it’s often deeply tied to inheritance, which can make it complicated and difficult to use. In Go, types are *com- posed* of smaller types, which is in contrast to traditional inheritance-based models.

In addition Go has a unique interface implementation that allows you to model behavior, rather than model types. You don’t need to declare that you’re implementing

Licensed to Mark Watson <nordickan@gmail.com>

**6** CHAPTER 1 ***Introducing Go***

an interface in Go; the compiler does the work of determining whether values of your types satisfy the interfaces you’re using. Many interfaces in Go’s standard library are very small, exposing only a few functions. In practice this takes some time to get used to, especially if you’ve been writing in object-oriented languages like Java.

TYPES ARE SIMPLE Go has built-in types like int and string as well as user-defined types. A typical user- defined type in Go will have typed fields to store data. If you’ve seen structs in C, Go’s user-defined types will look familiar and operate similarly. But types may also declare methods that operate on that data. Rather than building a long inheritance struc- ture—Client extends User extends Entity—Go developers build small types—Cus- tomer and Admin—and embed them into larger ones. Figure 1.4 demonstrates the difference between inheritance and composition.

Inheritance

Vehicle

Passenger vehicle

Truck

GO INTERFACES MODEL SMALL BEHAVIORS Interfaces allow you to express the behavior of a type. If a value of a type implements an interface, it means the value has a specific set of behaviors. You don’t even need to declare that you’re implementing an interface; you just need to write the implementa- tion. Other languages call this *duck typing*—if it quacks like a duck, then it can *be* a duck—and Go does it well. In Go, if your type implements the methods of an inter- face, a value of your type can be stored in a value of that interface type. No special dec- larations are required.

In a strictly object-oriented language like Java, interfaces are all-encompassing. You’re often required to think through a large inheritance chain before you’re able to even start writing code. Here’s an example of a Java interface:

interface User {

public void login(); public void logout(); }Implementing this interface in Java requires you to create a class that fulfills all of the promises made in the User interface and explicitly declare that you implement the interface. In contrast, a Go interface typically represents just a single action. One of the most common interfaces you’ll use in Go is io.Reader. The io.Reader interface

Composition

Drive

Passengers

Truck

Cargo

Carry cargo

Carry passengers

Drive Figure 1.4 Inheritance

versus composition

Licensed to Mark Watson <nordickan@gmail.com>

**7 *Hello, Go***

provides a simple way to declare that your type has data to be read in a way that other functions in the standard library understand. Here’s the definition:

type Reader interface {

Read(p []byte) (n int, err error) }To write a type that implements the io.Reader interface, you only need to implement a Read method that accepts a slice of bytes and returns an integer and possible error.

This is a radical departure from the interface systems used in other object-oriented programming languages. Go’s interfaces are smaller and more aligned with single actions. In practice, this allows significant advantages in code reuse and composability. You can implement an io.Reader on nearly any type that has data available, and then pass it to any Go function that knows how to read from io.Reader.

The entire networking library in Go is built using the io.Reader interface, because it allows it to separate the network implementation required for each different net- work operation from the functionality of your application. It makes interfaces fun, ele- gant, and flexible. That same io.Reader enables simple operations with files, buffers, sockets, and any other data source. Using a single interface allows you to operate on data efficiently, regardless of the source.

***1.1.4 Memory management***

Improper memory management causes applications to crash and leak memory, and even crash the operating system. Go has a modern garbage collector that does the hard work for you. In other systems languages, like C or C++, you need to allocate a piece of memory before you can use it, and then free it when you’re done. If you fail to do either of these correctly, you’ll have program crashes or memory leaks. It isn’t always easy to track a piece of memory when it’s no longer needed; threads and heavy concurrency make it even harder. When you write code with garbage collection in mind, Go’s garbage collection adds little overhead to program execution time, but reduces development effort significantly. Go takes the tedium out of programming and leaves the bean counting to the accountants.

***1.2 Hello, Go***

It’s much easier to get the feel of a programming language by seeing it in action. Let’s look at the traditional *Hello World!* application written in Go:

**Go programs are organized as packages.**

package main **The import statement allows you to use external code. The fmt package provided by the standard** import "fmt"

**library allows you to format and output data.**

func main(){ fmt.Println("Hello }

World!")

**The main function is what gets executed when you run your application—just like in C.**

Licensed to Mark Watson <nordickan@gmail.com>

**8** CHAPTER 1 ***Introducing Go***

This sample program prints a familiar phrase on your screen when you run it. But how should you run it? Without installing Go on your computer, you can use almost all that Go provides right from your web browser.

***1.2.1 Introducing the Go Playground***

The Go Playground allows you to edit and run Go code from your web browser. Fire up a web browser and navigate to http://play.golang.org. The code in the browser window is editable right on the screen (see figure 1.5). Click Run and see what happens!

You can even change the code to make the greeting text output in a different lan- guage. Go ahead and change the greeting inside the fmt.Println() function and hit Run again.

SHARING GO CODE Go developers use the Playground to share code ideas, test theories, and debug their code, as you soon will too. Every time you create a new application on the Playground, you can click Share to get a sharable URL that anyone else can open. Try this one: http://play.golang.org/p/EWIXicJdmz.

The Go Playground is the perfect way to demonstrate an idea to a coworker or friend who’s trying to learn something, or to solicit help. On the Go IRC channels, Slack group, mailing lists, and countless emails sent among Go developers, you’ll see Go Playground programs being created, modified, and shared.

***1.3 Summary***

▪ Go is modern, fast, and comes with a powerful standard library.

▪ Go has concurrency built-in.

▪ Go uses interfaces as the building blocks of code reuse.

Figure 1.5 The Go Playground

Licensed to Mark Watson <nordickan@gmail.com>

*Go quick-start*

***In this chapter***

▪ Reviewing a comprehensive Go program

▪ Declaring types, variables, functions, and methods

▪ Launching and synchronizing goroutines

▪ Writing generic code using interfaces

▪ Handling errors as normal program logic

Go has its own elegance and programming idioms that make the language produc- tive and fun to code in. The language designers set out to create a language that would let them be productive without losing access to the lower-level programming constructs they needed. This balance is achieved through a minimized set of key- words, built-in functions, and syntax. Go also provides a comprehensive standard library. The standard library provides all the core packages programmers need to build real-world web- and network-based programs.

To see this in action, we’ll review a complete Go program that implements func- tionality that can be found in many Go programs being developed today. The pro- gram pulls different data feeds from the web and compares the content against a search term. The content that matches is then displayed in the terminal window.

Licensed to Mark Watson <nordickan@gmail.com>

www.allitebooks.com

**9**

**10** CHAPTER 2 ***Go quick-start***

The program reads text files, makes web calls, and decodes both XML and JSON into struct type values, and it does all of this using Go concurrency to make things fast.

You can download and review the code in your favorite editor by navigating to the book repository for this chapter:

https://github.com/goinaction/code/tree/master/chapter2/sample

Don’t feel that you need to understand everything you read and review in this chapter the first, second, or even the third time. Though many of the programming concepts you know today can be applied when learning Go, Go also has its unique idioms and style. If you can liberate yourself from your current programming language and look at Go with a fresh set of eyes and a clear mind, you’ll find it easier to understand and appreciate, and you’ll see Go’s elegance.

***2.1 Program architecture***

Before we dive into the code, let’s review the architecture behind the program (shown in figure 2.1) and see how searching all the different feeds is accomplished.

Main goroutine

Set of feeds

Retrieve

to search

feeds

Perform

Search goroutines search

Interface

Send matcher

result

Track results

Track results goroutine

Display results Shutdown Wait all

Goroutine Channel

The program is broken into several distinct steps that run across many different goroutines. We’ll explore the code as it flows from the main goroutine into the search- ing and tracking goroutines and then back to the main goroutine. To start, here’s the structure of the project.

cd $GOPATH/src/github.com/goinaction/code/chapter2

- sample

- datadata.json -- Contains a list of data feeds - matchers

Report complete

Figure 2.1 The flow of the program architecture

Listing 2.1 Project structure for the application

Licensed to Mark Watson <nordickan@gmail.com>

**11 *Main package***

rss.go -- Matcher for searching rss feeds - search

default.go -- Default matcher for searching data feed.go -- Support for reading the json data file match.go -- Interface support for using different matchers search.go -- Main program logic for performing search main.go -- Programs entry point

The code is organized into these four folders, which are listed in alphabetical order. The data folder contains a JSON document of data feeds the program will retrieve and process to match the search term. The matchers folder contains the code for the different types of feeds the program supports. Currently the program only supports one matcher that processes RSS type feeds. The search folder contains the business logic for using the different matchers to search content. Finally we have the parent folder, sample, which contains the main.go code file, which is the entry point for the program. Now that you’ve seen where all the code for the program is, you can begin to explore and understand how the program works. Let’s start with the entry point for the program.

***2.2 Main package***

The program’s entry point can be found in the main.go code file. Even though there are only 21 lines of code, there are a few things going on that we have to mention.

Listing 2.2 main.go

01 package main 0203 import ( 04 "log" 05 "os" 0607 \_ "github.com/goinaction/code/chapter2/sample/matchers" 08 "github.com/goinaction/code/chapter2/sample/search" 09 ) 1011 // init is called prior to main. 12 func init() { 13 // Change the device for logging to stdout. 14 log.SetOutput(os.Stdout) 15 } 1617 // main is the entry point for the program. 18 func main() { 19 // Perform the search for the specified term. 20 search.Run("president") 21 }

Every Go program that produces an executable has two distinct features. One of those features can be found on line 18. There you can see the function main declared. For

Licensed to Mark Watson <nordickan@gmail.com>

**12** CHAPTER 2 ***Go quick-start***

the build tools to produce an executable, the function main must be declared, and it becomes the entry point for the program. The second feature can be found on line 01 of program.

Listing 2.3 main.go: line 01

01 package main

You can see the function main is located in a package called main. If your main func- tion doesn’t exist in package main, the build tools won’t produce an executable.

Every code file in Go belongs to a package, and main.go is no exception. We’ll go into much more detail about packages in chapter 3, because packages are an impor- tant feature of Go. For now, understand that packages define a unit of compiled code and their names help provide a level of indirection to the identifiers that are declared inside of them, just like a namespace. This makes it possible to distinguish identifiers that are declared with exactly the same name in the different packages you import.

Now turn your attention to lines 03 through 09 of the main.go code file, which declares imports.

Listing 2.4 main.go: lines 03–09

03 import ( 04 "log" 05 "os" 0607 \_ "github.com/goinaction/code/chapter2/sample/matchers" 08 "github.com/goinaction/code/chapter2/sample/search" 09 )

Imports are just that: they import code and give you access to identifiers such as types, functions, constants, and interfaces. In our case, the code in the main.go code file can now reference the Run function from the search package, thanks to the import on line 08. On lines 04 and 05, we import code from the standard library for the log and os packages.

All code files in a folder must use the same package name, and it’s common prac- tice to name the package after the folder. As stated before, a package defines a unit of compiled code, and each unit of code represents a package. If you quickly look back at listing 2.1, you’ll see how we have a folder in this project called search that matches the import path on line 08.

You may have noticed that on line 07 we import the matchers package and use the blank identifier before listing out the import path.

Listing 2.5 main.go: line 07

07 \_ "github.com/goinaction/code/chapter2/sample/matchers"

Licensed to Mark Watson <nordickan@gmail.com>

**13 *Search package***

This is a technique in Go to allow initialization from a package to occur, even if you don’t directly use any identifiers from the package. To make your programs more readable, the Go compiler won’t let you declare a package to be imported if it’s not used. The blank identifier allows the compiler to accept the import and call any init functions that can be found in the different code files within that package. For our program, this is required because the rss.go code file in the matchers package con- tains an init function to register the RSS matcher for use. We’ll come back to how all this works later.

The main.go code file also has an init function that’s declared on lines 12 through 15.

Listing 2.6 main.go: lines 11–15

11 // init is called prior to main. 12 func init() { 13 // Change the device for logging to stdout. 14 log.SetOutput(os.Stdout) 15 }

All init functions in any code file that are part of the program will get called before the main function. This init function sets the logger from the standard library to write to the stdout device. By default, the logger is set to write to the stderr device. In chapter 7 we’ll talk more about the log package and other important packages from the standard library.

Finally, let’s look at the one statement that the main function performs on line 20.

Listing 2.7 main.go: lines 19–20

19 // Perform the search for the specified term. 20 search.Run("president")

Here you see a call to the Run function that belongs to the search package. This func- tion contains the core business logic for the program, which requires a string for the search term. Once the Run function returns, the program will terminate. Now we can look at the code that belongs to the search package.

***2.3 Search package***

The search package contains the framework and business logic for the program. The package is organized into four different code files, each with a unique responsibility. As we continue to follow the logic of the program, we’ll explore each of these differ- ent code files.

Let’s briefly talk about what a matcher is, since the entire program revolves around the execution of matchers. A matcher in our program is a value that contains specific intelligence for processing a feed type. In our program we have two matchers. The framework implements a default matcher that has no intelligence, and in the matchers

Licensed to Mark Watson <nordickan@gmail.com>

**14** CHAPTER 2 ***Go quick-start***

package we have an implementation of an RSS matcher. The RSS matcher knows how to get, read, and search RSS feeds. Later on we could extend the program to use matchers that could read JSON documents or CSV files. We’ll talk more about how to implement matchers later.

***2.3.1 search.go***

Following are the first nine lines of code that can be found inside the search.go code file. This is the code file where the Run function is located.

Listing 2.8 search/search.go: lines 01–09

01 package search 0203 import ( 04 "log" 05 "sync" 06 ) 0708 // A map of registered matchers for searching. 09 var matchers = make(map[string]Matcher)

As you’ll see, each code file will contain the keyword package at the top with a name for the package. Each code file in the search folder will contain search for the pack- age name. The lines from 03 through 06 import the log and sync packages from the standard library.

When you import code from the standard library, you only need to reference the name of the package, unlike when you import code from outside of the standard library. The compiler will always look for the packages you import at the locations ref- erenced by the GOROOT and GOPATH environment variables.

Listing 2.9 **GOROOT** and **GOPATH** environmental variables

GOROOT="/Users/me/go" GOPATH="/Users/me/spaces/go/projects"

The log package provides support for logging messages to the stdout, stderr, or even custom devices. The sync package provides support for synchronizing goroutines, which is required by our program. On line 09 you’ll see our first variable declaration.

Listing 2.10 search/search.go: lines 08–09

08 // A map of registered matchers for searching. 09 var matchers = make(map[string]Matcher)

This variable is located outside the scope of any function and so is considered a package-level variable. The variable is declared using the keyword var and is declared as a map of Matcher type values with a key of type string. The declaration for the

Licensed to Mark Watson <nordickan@gmail.com>

**15 *Search package***

Matcher type can be found in the match.go code file, and we’ll describe the purpose of this type later. There’s another important aspect of this variable declaration: the name of the variable matchers starts with a lowercase letter.

In Go, identifiers are either exported or unexported from a package. An exported identifier can be directly accessed by code in other packages when the respective package is imported. These identifiers start with a capital letter. Unexported identifi- ers start with a lowercase letter and can’t be directly accessed by code in other pack- ages. But just because an identifier is unexported, it doesn’t mean other packages can’t indirectly access these identifiers. As an example, a function can return a value of an unexported type and this value is accessible by any calling function, even if the calling function has been declared in a different package.

This variable declaration also contains an initialization of the variable via the assignment operator and a special built-in function called make.

Listing 2.11 Making a map

make(map[string]Matcher)

A map is a reference type that you’re required to make in Go. If you don’t make the map first and assign it to your variable, you’ll receive errors when you try to use the map variable. This is because the zero value for a map variable is nil. In chapter 4 we’ll go into greater detail about maps.

In Go, all variables are initialized to their zero value. For numeric types, that value is 0; for strings it’s an empty string; for Booleans it’s false; and for pointers, the zero value is nil. When it comes to reference types, there are underlying data structures that are initialized to their zero values. But variables declared as a reference type set to their zero value will return the value of nil.

Now let’s walk through the Run function that’s called by the main function, which you saw earlier.

Listing 2.12 search/search.go: lines 11–57

11 // Run performs the search logic. 12 func Run(searchTerm string) { 13 // Retrieve the list of feeds to search through. 14 feeds, err := RetrieveFeeds() 15 if err != nil { 16 log.Fatal(err) 17 } 1819 // Create a unbuffered channel to receive match results. 20 results := make(chan \*Result) 2122 // Setup a wait group so we can process all the feeds. 23 var waitGroup sync.WaitGroup 2425 // Set the number of goroutines we need to wait for while

Licensed to Mark Watson <nordickan@gmail.com>

**16** CHAPTER 2 ***Go quick-start***

26 // they process the individual feeds. 27 waitGroup.Add(len(feeds)) 2829 // Launch a goroutine for each feed to find the results. 30 for \_, feed := range feeds { 31 // Retrieve a matcher for the search. 32 matcher, exists := matchers[feed.Type] 33 if !exists { 34 matcher = matchers["default"] 35 } 3637 // Launch the goroutine to perform the search. 38 go func(matcher Matcher, feed \*Feed) { 39 Match(matcher, feed, searchTerm, results) 40 waitGroup.Done() 41 }(matcher, feed) 42 } 4344 // Launch a goroutine to monitor when all the work is done. 45 go func() { 46 // Wait for everything to be processed. 47 waitGroup.Wait() 4849 // Close the channel to signal to the Display 50 // function that we can exit the program. 51 close(results) 52 }() 5354 // Start displaying results as they are available and 55 // return after the final result is displayed. 56 Display(results) 57 }

The Run function contains the main control logic for the program. It’s a good repre- sentation of how Go programs can be structured to handle the launching and syn- chronization of goroutines that run concurrently. Let’s walk through the logic section by section, and then explore the other code files that lend their support.

Let’s review how the Run function is declared.

Listing 2.13 search/search.go: lines 11–12

11 // Run performs the search logic. 12 func Run(searchTerm string) {

To declare a function in Go, use the keyword func followed by the function name, any parameters, and then any return values. In the case of Run, you have a single parame- ter called searchTerm of type string. The term the program will search against is passed into the Run function, and if you look at the main function again, you can see that exchange.

Licensed to Mark Watson <nordickan@gmail.com>

**17 *Search package***

Listing 2.14 main.go: lines 17–21

17 // main is the entry point for the program. 18 func main() { 19 // Perform the search for the specified term. 20 search.Run("president") 21 }

The first thing that the Run function does is retrieve a list of data feeds. These feeds are used to pull content from the internet that is then matched against the specified search term.

Listing 2.15 search/search.go: lines 13–17

13 // Retrieve the list of feeds to search through. 14 feeds, err := RetrieveFeeds() 15 if err != nil { 16 log.Fatal(err) 17 }

There are a few important concepts here that we need to go through. You can see on line 14 that we make a function call to the function RetrieveFeeds. This function belongs to the search package and returns two values. The first return value is a slice of Feed type values. A slice is a reference type that implements a dynamic array. You use slices in Go to work with lists of data. Chapter 4 goes into greater detail about slices.

The second return value is an error. On line 15, the error value is evaluated for errors, and if an error did occur, the function Fatal from the log package is called. The Fatal function accepts an error value and will log to the terminal window before terminating the program.

Though not unique to Go, you can see that our functions can have multiple return values. It’s common to declare functions that return a value and an error value just like the RetrieveFeeds function. If an error occurs, never trust the other values being returned from the function. They should always be ignored, or else you run the risk of the code generating more errors or panics.

Let’s take a closer look at how the values being returned from the function are being assigned to variables.

Listing 2.16 search/search.go: lines 13–14

13 // Retrieve the list of feeds to search through. 14 feeds, err := RetrieveFeeds()

Here you see the use of the short variable declaration operator (:=). This operator is used to both declare and initialize variables at the same time. The type of each value being returned is used by the compiler to determine the type for each variable, respectively. The short variable declaration operator is just a shortcut to streamline

Licensed to Mark Watson <nordickan@gmail.com>

**18** CHAPTER 2 ***Go quick-start***

your code and make the code more readable. The variable it declares is no different than any other variable you may declare when using the keyword var.

Now that we have our list of data feeds, we can move on to the next line of code.

Listing 2.17 search/search.go: lines 19–20

19 // Create a unbuffered channel to receive match results. 20 results := make(chan \*Result)

On line 20, we use the built-in function make to create an unbuffered channel. We use the short variable declaration operator to declare and initialize the channel variable with the call to make. A good rule of thumb when declaring variables is to use the key- word var when declaring variables that will be initialized to their zero value, and to use the short variable declaration operator when you’re providing extra initialization or making a function call.

Channels are also a reference type in Go like maps and slices, but channels imple- ment a queue of typed values that are used to communicate data between goroutines. Channels provide inherent synchronization mechanisms to make communication safe. In chapter 6 we’ll go into more details about channels and goroutines.

The next two lines of code are used later to prevent the program from terminating before all the search processing is complete.

Listing 2.18 search/search.go: lines 22–27

22 // Setup a wait group so we can process all the feeds. 23 var waitGroup sync.WaitGroup 2425 // Set the number of goroutines we need to wait for while 26 // they process the individual feeds. 27 waitGroup.Add(len(feeds))

In Go, once the main function returns, the program terminates. Any goroutines that were launched and are still running at this time will also be terminated by the Go run- time. When you write concurrent programs, it’s best to cleanly terminate any gorou- tines that were launched prior to letting the main function return. Writing programs that can cleanly start and shut down helps reduce bugs and prevents resources from corruption.

Our program is using a WaitGroup from the sync package to track all the gorou- tines we’re going to launch. A WaitGroup is a great way to track when a goroutine is finished performing its work. A WaitGroup is a counting semaphore, and we’ll use it to count off goroutines as they finish their work.

On line 23 we declare a variable of type WaitGroup from the sync package. Then on line 27 we set the value of the WaitGroup variable to match the number of gorou- tines we’re going to launch. As you’ll soon see, we’ll process each feed concurrently with its own goroutine. As each goroutine completes its work, it will decrement the

Licensed to Mark Watson <nordickan@gmail.com>

**19 *Search package***

count of the WaitGroup variable, and once the variable gets to zero, we’ll know all the work is done.

Next let’s look at the code that launches these goroutines for each feed.

Listing 2.19 search/search.go: lines 29–42

29 // Launch a goroutine for each feed to find the results. 30 for \_, feed := range feeds { 31 // Retrieve a matcher for the search. 32 matcher, exists := matchers[feed.Type] 33 if !exists { 34 matcher = matchers["default"] 35 } 3637 // Launch the goroutine to perform the search. 38 go func(matcher Matcher, feed \*Feed) { 39 Match(matcher, feed, searchTerm, results) 40 waitGroup.Done() 41 }(matcher, feed) 42 }

The code for lines 30 through 42 iterate through the list of data feeds we retrieved earlier and launch a goroutine for each one. To iterate over the slice of feeds, we use the keywords for range. The keyword range can be used with arrays, strings, slices, maps, and channels. When we use for range to iterate over a slice, we get two values back on each iteration. The first is the index position of the element we’re iterating over, and the second is a copy of the value in that element.

If you look closer at the for range statement on line 30, you’ll see the use of the blank identifier again.

Listing 2.20 search/search.go: lines 29–30

29 // Launch a goroutine for each feed to find the results. 30 for \_, feed := range feeds {

This is the second time you see the blank identifier being used. You first saw it in main.go when we imported the matchers package. Now it’s being used as a substitu- tion for the variable that would be assigned to the index value for the range call. When you have a function that returns multiple values, and you don’t have a need for one, you can use the blank identifier to ignore those values. In our case with this range, we won’t be using the index value, so the blank identifier allows us to ignore it.

The first thing we do in the loop is check the map for a Matcher value that can be used to process a feed of the specific feed type.

Listing 2.21 search/search.go: lines 31–35

31 // Retrieve a matcher for the search. 32 matcher, exists := matchers[feed.Type]

Licensed to Mark Watson <nordickan@gmail.com>

**20** CHAPTER 2 ***Go quick-start***

33 if !exists { 34 matcher = matchers["default"] 35 }

We haven’t talked about how this map gets its values yet. You’ll see later on how the program initializes itself and populates this map. On line 32 we check the map for a key that matches the feed type. When looking up a key in a map, you have two options: you can assign a single variable or two variables for the lookup call. The first variable is always the value returned for the key lookup, and the second value, if speci- fied, is a Boolean flag that reports whether the key exists or not. When a key doesn’t exist, the map will return the zero value for the type of value being stored in the map. When the key does exist, the map will return a copy of the value for that key.

On line 33 we check whether the key was located in the map, and if it’s not, we assign the default matcher to be used. This allows the program to function without causing any issues or interruption for feeds that the program currently doesn’t sup- port. Then we launch a goroutine to perform the search.

Listing 2.22 search/search.go: lines 37–41

37 // Launch the goroutine to perform the search. 38 go func(matcher Matcher, feed \*Feed) { 39 Match(matcher, feed, searchTerm, results) 40 waitGroup.Done() 41 }(matcher, feed)

In chapter 6 we’ll go into more detail about goroutines, but for now a *goroutine* is a function that’s launched to run independently from other functions in the program. Use the keyword go to launch and schedule goroutines to run concurrently. On line 38 we use the keyword go to launch an anonymous function as a goroutine. An *anony- mous function* is a function that’s declared without a name. In our for range loop, we launch an anonymous function as a goroutine for each feed. This allows each feed to be processed independently in a concurrent fashion.

Anonymous functions can take parameters, which we declare for this anonymous function. On line 38 we declare the anonymous function to accept a value of type Matcher and the address of a value of type Feed. This means the variable feed is a *pointer variable*. Pointer variables are great for sharing variables between functions. They allow functions to access and change the state of a variable that was declared within the scope of a different function and possibly a different goroutine.

On line 41 the values of the matcher and feed variables are being passed into the anonymous function. In Go, all variables are passed by value. Since the value of a pointer variable is the address to the memory being pointed to, passing pointer vari- ables between functions is still considered a pass by value.

On lines 39 and 40 you see the work each goroutine is performing.

Licensed to Mark Watson <nordickan@gmail.com>

**21 *Search package***

Listing 2.23 search/search.go: lines 39–40

39 Match(matcher, feed, searchTerm, results) 40 waitGroup.Done()

The first thing the goroutine does is call a function called Match, which can be found in the match.go code file. The Match function takes a value of type Matcher, a pointer to a value of type Feed, the search term, and the channel where the results are written to. We’ll look at the internals of this function later, but for now it’s enough to know that Match will search the feed and output matches to the results channel.

Once the function call to Match completes, we execute the code on line 40, which is to decrement the WaitGroup count. Once every goroutine finishes calling the Match function and the Done method, the program will know every feed has been processed. There’s something else interesting about the method call to Done: the WaitGroup value was never passed into the anonymous function as a parameter, yet the anony- mous function has access to it.

Go supports closures and you’re seeing this in action. In fact, the searchTerm and results variables are also being accessed by the anonymous function via closures. Thanks to closures, the function can access those variables directly without the need to pass them in as parameters. The anonymous function isn’t given a copy of these vari- ables; it has direct access to the same variables declared in the scope of the outer func- tion. This is the reason why we don’t use closures for the matcher and feed variables.

Listing 2.24 search/search.go: lines 29–32

29 // Launch a goroutine for each feed to find the results. 30 for \_, feed := range feeds { 31 // Retrieve a matcher for the search. 32 matcher, exists := matchers[feed.Type]

The values of the feed and matcher variables are changing with each iteration of the loop, as you can see on lines 30 and 32. If we used closures for these variables, as the values of these variables changed in the outer function, those changes would be reflected in the anonymous function. All the goroutines would be sharing the same variables as the outer function thanks to closures. Unless we passed these values in as function parameters, most of the goroutines would end up processing the same feed using the same matcher—most likely the last one in the feeds slice.

With all the search goroutines working, sending results on the results channel and decrementing the waitGroup counter, we need a way to display those results and keep the main function alive until all the processing is done.

Listing 2.25 search/search.go: lines 44–57

44 // Launch a goroutine to monitor when all the work is done. 45 go func() { 46 // Wait for everything to be processed.

Licensed to Mark Watson <nordickan@gmail.com>

**22** CHAPTER 2 ***Go quick-start***

47 waitGroup.Wait() 4849 // Close the channel to signal to the Display 50 // function that we can exit the program. 51 close(results) 52 }() 5354 // Start displaying results as they are available and 55 // return after the final result is displayed. 56 Display(results) 57 }

The code between lines 45 and 56 is tricky to explain until we dive deeper into some of the other code in the search package. For now let’s describe what we see and come back to it later to understand the mechanics. On lines 45 through 52 we launch yet another anonymous function as a goroutine. This anonymous function takes no parameters and uses closures to access both the WaitGroup and results variables. This goroutine calls the method Wait on the WaitGroup value, which is causing the goroutine to block until the count for the WaitGroup hits zero. Once that happens, the goroutine calls the built-in function close on the channel, which as you’ll see causes the program to terminate.

The final piece of code in the Run function is on line 56. This is a call to the Display function, which can be found in the match.go code file. Once this function returns, the program terminates. This doesn’t happen until all the results in the chan- nel are processed.

***2.3.2 feed.go***

Now that you’ve seen the Run function, let’s look at the code behind the function call to RetrieveFeeds on line 14 of the search.go code file. This function reads the data.json file and returns the slice of data feeds. These feeds drive the content that will be searched by the different matchers. Here are the first eight lines of code that can be found inside the feed.go code file.

Listing 2.26 feed.go: lines 01–08

01 package search 0203 import ( 04 "encoding/json" 05 "os" 06 ) 0708 const dataFile = "data/data.json"

This code file exists in the search folder, and on line 01 the code file is declared to be in package search. You can see that on lines 03 through 06 we import two packages from the standard library. The json package provides support for encoding and decoding JSON and the os package provides support for accessing operating system functionality like reading files.

Licensed to Mark Watson <nordickan@gmail.com>

**23 *Search package***

You may have noticed that to import the json package, we needed to specify a path that includes the encoding folder. Regardless of the path we specify, the name of the package is json. The physical location of the package from within the standard library doesn’t change this fact. As we access functionality from the json package, we’ll use just the name json.

On line 08 we declare a constant named dataFile, which is assigned a string that specifies the relative path to the data file on disk. Since the Go compiler can deduce the type from the value on the right side of the assignment operator, specifying the type when declaring the constant is unnecessary. We also use a lowercase letter for the name of the constant, which means this constant is unexported and can only be directly accessed by code within the search package.

Next let’s look at a portion of the data.json data file.

Listing 2.27 data.json

[

{

"site" : "npr", "link" : "http://www.npr.org/rss/rss.php?id=1001", "type" : "rss" },{

"site" : "cnn", "link" : "http://rss.cnn.com/rss/cnn\_world.rss", "type" : "rss" },{

"site" : "foxnews", "link" : "http://feeds.foxnews.com/foxnews/world?format=xml", "type" : "rss" },{

"site" : "nbcnews", "link" : "http://feeds.nbcnews.com/feeds/topstories", "type" : "rss" } ]The actual data file contains more than four data feeds, but listing 2.27 shows a valid version of the data file. The data file contains an array of JSON documents. Each docu- ment in the data file provides a name of the site we’re getting the data from, a link to the data, and the type of data we expect to receive.

These documents need to be decoded into a slice of struct types so we can use this data in our program. Let’s look at the struct type that will be used to decode this data file.

Listing 2.28 feed.go: lines 10–15

10 // Feed contains information we need to process a feed. 11 type Feed struct { 12 Name string `json:"site"`

Licensed to Mark Watson <nordickan@gmail.com>

**24** CHAPTER 2 ***Go quick-start***

13 URI string `json:"link"` 14 Type string `json:"type"` 15 }

On lines 11 through 15 we declare a struct type named Feed, which is an exported type. This type is declared with three fields, each of which are strings that match the fields for each document in the data file. If you look at each field declaration, tags have been included to provide the metadata that the JSON decoding function needs to create the slice of Feed type values. Each tag maps a field name in the struct type to a field name in the document.

Now we can review the RetrieveFeeds function that we called on line 14 in the search.go code file. This is the function that reads the data file and decodes every doc- ument into a slice of Feed type values.

Listing 2.29 feed.go: lines 17–36

17 // RetrieveFeeds reads and unmarshals the feed data file. 18 func RetrieveFeeds() ([]\*Feed, error) { 19 // Open the file. 20 file, err := os.Open(dataFile) 21 if err != nil { 22 return nil, err 23 } 2425 // Schedule the file to be closed once 26 // the function returns. 27 defer file.Close() 2829 // Decode the file into a slice of pointers 30 // to Feed values. 31 var feeds []\*Feed 32 err = json.NewDecoder(file).Decode(&feeds) 3334 // We don't need to check for errors, the caller can do this. 35 return feeds, err 36 }

Let’s start with the declaration of the function on line 18. The function takes no parameters and returns two values. The first return value is a slice of pointers to Feed type values. The second return value is an error value that reports back if the function call was successful. As you’ll continue to see, returning error values is common prac- tice in this code example and throughout the standard library.

Now let’s look at lines 20 through 23, where we use the os package to open the data file. The call to the Open method takes the relative path to our data file and returns two values. The first return value is a pointer to a value of type File, and the second return value is an error to check if the call to Open was successful. Immediately on line 21 we check the error value and return the error if we did have a problem opening the file.

Licensed to Mark Watson <nordickan@gmail.com>

**25 *Search package***

If we’re successful in opening the file, we then move to line 27. Here you see the use of the keyword defer.

Listing 2.30 feed.go: lines 25–27

25 // Schedule the file to be closed once 26 // the function returns. 27 defer file.Close()

The keyword defer is used to schedule a function call to be executed right after a function returns. It’s our responsibility to close the file once we’re done with it. By using the keyword defer to schedule the call to the close method, we can guarantee that the method will be called. This will happen even if the function panics and termi- nates unexpectedly. The keyword defer lets us write this statement close to where the opening of the file occurs, which helps with readability and reducing bugs.

Now we can review the final lines of code in the function. Let’s look at lines 31 through 35.

Listing 2.31 feed.go: lines 29–36

29 // Decode the file into a slice of pointers 30 // to Feed values. 31 var feeds []\*Feed 32 err = json.NewDecoder(file).Decode(&feeds) 3334 // We don't need to check for errors, the caller can do this. 35 return feeds, err 36 }

On line 31 we declare a nil slice named feeds that contains pointers to Feed type val- ues. Then on line 32 we make a call to the Decode method off the value returned by the NewDecoder function from the json package. The NewDecoder function takes the file handle we created from the method call to Open and returns a pointer to a value of type Decoder. From that value we call the Decode method, passing the address to the slice. The Decode method then decodes the data file and populates our slice with a set of Feed type values.

The Decode method can accept any type of value thanks to its declaration.

Listing 2.32 Using the empty interface

func (dec \*Decoder) Decode(v interface{}) error

The parameter for the Decode method accepts a value of type interface{}. This is a special type in Go and works with the reflection support that can be found in the reflect package. In chapter 9 we’ll go into more detail about reflection and how this method works.

The last line of code on line 35 returns the slice and error values back to the call- ing function. In this case there’s no need for the function to check the error value

Licensed to Mark Watson <nordickan@gmail.com>

**26** CHAPTER 2 ***Go quick-start***

after the call to Decode. The function is complete and the calling function can check the error value and determine what to do next.

Now it’s time to see how the search code supports different types of feed imple- mentations by reviewing the matcher code.

***2.3.3 match.go/default.go***

The match.go code file contains the support for creating different types of matchers that can be used by the search Run function. Let’s go back and look at the code from the Run function that executes the search using the different types of matchers.

29 // Launch a goroutine for each feed to find the results. 30 for \_, feed := range feeds { 31 // Retrieve a matcher for the search. 32 matcher, exists := matchers[feed.Type] 33 if !exists { 34 matcher = matchers["default"] 35 } 3637 // Launch the goroutine to perform the search. 38 go func(matcher Matcher, feed \*Feed) { 39 Match(matcher, feed, searchTerm, results) 40 waitGroup.Done() 41 }(matcher, feed) 42 }

The code on line 32 looks up a matcher value based on the feed type; that value is then used to process a search against that specific feed. Then on line 38 through 41, a goroutine is launched for that matcher and feed value. The key to making this code work is the ability of this framework code to use an interface type to capture and call into the specific implementation for each matcher value. This allows the code to han- dle different types of matcher values in a consistent and generic way. Let’s look at the code in match.go and see how we’re able to implement this functionality.

Here are the first 17 lines of code for match.go.

01 package search 0203 import ( 04 "log" 05 ) 0607 // Result contains the result of a search. 08 type Result struct { 09 Field string 10 Content string 11 } 12

Listing 2.33 search/search.go : lines 29 - 42

Listing 2.34 search/match.go: lines 01–17

Licensed to Mark Watson <nordickan@gmail.com>

**27 *Search package***

13 // Matcher defines the behavior required by types that want 14 // to implement a new search type. 15 type Matcher interface { 16 Search(feed \*Feed, searchTerm string) ([]\*Result, error) 17 }

Let’s jump to lines 15 through 17 and look at the declaration of the interface type named Matcher. Up until now we’ve only been declaring struct types, but here you see code that’s declaring an interface type. We’ll get into a lot more detail about inter- faces in chapter 5, but for now know that interfaces declare behavior that’s required to be implemented by struct or named types to satisfy the interface. The behavior of an interface is defined by the methods that are declared within the interface type.

In the case of the Matcher interface, there’s only one method declared, Search, which takes a pointer to a value of type Feed and a search term of type string. The method also returns two values: a slice of pointers to values of type Result and an error value. The Result type is declared on lines 08 through 11.

You follow a naming convention in Go when naming interfaces. If the interface type contains only one method, the name of the interface ends with the *er* suffix. This is the exact case for our interface, so the name of the interface is Matcher. When mul- tiple methods are declared within an interface type, the name of the interface should relate to its general behavior.

For a user-defined type to implement an interface, the type in question needs to implement all the methods that are declared within that interface type. Let’s switch to the default.go code file and see how the default matcher implements the Matcher interface.

Listing 2.35 search/default.go: lines 01–15

01 package search 0203 // defaultMatcher implements the default matcher. 04 type defaultMatcher struct{} 0506 // init registers the default matcher with the program. 07 func init() { 08 var matcher defaultMatcher 09 Register("default", matcher) 10 } 1112 // Search implements the behavior for the default matcher. 13 func (m defaultMatcher) Search(feed \*Feed, searchTerm string)

([]\*Result, error) { 14 return nil, nil 15 }

On line 04 we declare a struct type named defaultMatcher using an empty struct. An empty struct allocates zero bytes when values of this type are created. They’re great when you need a type but not any state. For the default matcher, we don’t need to maintain any state; we only need to implement the interface.

Licensed to Mark Watson <nordickan@gmail.com>

**28** CHAPTER 2 ***Go quick-start***

On lines 13 through 15 you see the implementation of the Matcher interface by the defaultMatcher type. The implementation of the interface method Search just returns nil for both return values. Other implementations, such as the implementa- tion for the RSS matcher, will implement the specific business rules for processing searches in their version of this method.

The declaration of the Search method is declared with a value receiver of type defaultMatcher.

Listing 2.36 search/default.go: line 13

13 func (m defaultMatcher) Search

The use of a receiver with any function declaration declares a method that’s bound to the specified receiver type. In our case, the declaration of the Search method is now bound to values of type defaultMatcher. This means we can call the method Search from values and pointers of type defaultMatcher. Whether we use a value or pointer of the receiver type to make the method call, the compiler will reference or derefer- ence the value if necessary to support the call.

Listing 2.37 Example of method calls

// Method declared with a value receiver of type defaultMatcher func (m defaultMatcher) Search(feed \*Feed, searchTerm string)

// Declare a pointer of type defaultMatch dm := new(defaultMatch)

// The compiler will dereference the dm pointer to make the call dm.Search(feed, "test")

// Method declared with a pointer receiver of type defaultMatcher func (m \*defaultMatcher) Search(feed \*Feed, searchTerm string)

// Declare a value of type defaultMatch var dm defaultMatch

// The compiler will reference the dm value to make the call dm.Search(feed, "test")

It’s best practice to declare methods using pointer receivers, since many of the methods you implement need to manipulate the state of the value being used to make the method call. In the case of the defaultMatcher type, we want to use a value receiver because creating values of type defaultMatcher result in values of zero allo- cation. Using a pointer makes no sense since there’s no state to be manipulated.

Unlike when you call methods directly from values and pointers, when you call a method via an interface type value, the rules are different. Methods declared with pointer receivers can only be called by interface type values that contain pointers. Methods declared with value receivers can be called by interface type values that con- tain both values and pointers.

Licensed to Mark Watson <nordickan@gmail.com>

**29 *Search package***

Listing 2.38 Example of interface method call restrictions

// Method declared with a pointer receiver of type defaultMatcher func (m \*defaultMatcher) Search(feed \*Feed, searchTerm string)

// Call the method via an interface type value var dm defaultMatcher var matcher Matcher = dm // Assign value to interface type matcher.Search(feed, "test") // Call interface method with value

> go build cannot use dm (type defaultMatcher) as type Matcher in assignment

// Method declared with a value receiver of type defaultMatcher func (m defaultMatcher) Search(feed \*Feed, searchTerm string)

// Call the method via an interface type value var dm defaultMatcher var matcher Matcher = &dm // Assign pointer to interface type matcher.Search(feed, "test") // Call interface method with pointer

> go build Build Successful

There’s nothing else that the defaultMatcher type needs to do to implement the interface. From this point forward, values and pointers of type defaultMatcher satisfy the interface and can be used as values of type Matcher. That’s the key to making this work. Values and pointers of type defaultMatcher are now also values of type Matcher and can be assigned or passed to functions accepting values of type Matcher.

Let’s look at the implementation of the Match function declared in the match.go code file. This is the function called by the Run function on line 39 in the search.go code file.

Listing 2.39 search/match.go: lines 19–33

19 // Match is launched as a goroutine for each individual feed to run 20 // searches concurrently. 21 func Match(matcher Matcher, feed \*Feed, searchTerm string,

results chan<- \*Result) { 22 // Perform the search against the specified matcher. 23 searchResults, err := matcher.Search(feed, searchTerm) 24 if err != nil { 25 log.Println(err) 26 return 27 } 2829 // Write the results to the channel. 30 for \_, result := range searchResults { 31 results <- result 32 } 33 }

This is the function that performs the actual search using values or pointers that implement the Matcher interface. This function accepts values of type Matcher as the

Licensed to Mark Watson <nordickan@gmail.com>

www.allitebooks.com

**30** CHAPTER 2 ***Go quick-start***

first parameter. Only values or pointers that implement the Matcher interface will be accepted for this parameter. Since the defaultMatcher type now implements the interface declared with a value receiver, values or pointers of type defaultMatcher can be passed into this function.

On line 23, the Search method is called from the Matcher type value that was passed into the function. Here the specific implementation of the Search method for the value assigned to the Matcher variable is executed. Once the Search method returns, the error value on line 24 is checked for an error. If there’s an error, the func- tion writes the error to the log and returns. If the search doesn’t return an error and there are results, the results are written to the channel so that they can be picked up by the main function that’s listening on that channel.

The final piece of code in match.go is the Display function that’s called by the main function on line 56. This is the function preventing the program from terminat- ing until all the results from the search goroutines are received and logged.

35 // Display writes results to the terminal window as they 36 // are received by the individual goroutines. 37 func Display(results chan \*Result) { 38 // The channel blocks until a result is written to the channel. 39 // Once the channel is closed the for loop terminates. 40 for result := range results { 41 fmt.Printf("%s:\n%s\n\n", result.Field, result.Content) 42 } 43 }

A bit of channel magic allows this function to process all of the results before return- ing. It’s based on how channels and the keyword range behaves when a channel is closed. Let’s briefly look at the code in the Run function again that closes the results channel and calls the Display function.

44 // Launch a goroutine to monitor when all the work is done. 45 go func() { 46 // Wait for everything to be processed. 47 waitGroup.Wait() 4849 // Close the channel to signal to the Display 50 // function that we can exit the program. 51 close(results) 52 }() 5354 // Start displaying results as they are available and 55 // return after the final result is displayed. 56 Display(results) 57 }Listing 2.40 search/match.go: lines 35–43 Listing 2.41 search/search.go: lines 44–57

Licensed to Mark Watson <nordickan@gmail.com>

**31 *Search package***

The goroutine on lines 45 through 52 waits on the waitGroup for all the search gorou- tines to call the Done method. Once the last search goroutine calls Done, the Wait method returns, and then the code on line 51 closes the results channel. Once the channel is closed, the goroutine terminates and is no more.

You saw on lines 30 through 32 in the match.go code file where the search results were being written to the channel.

Listing 2.42 search/match.go: lines 29–32

29 // Write the results to the channel. 30 for \_, result := range searchResults { 31 results <- result 32 }

If we look back at the for range loop on lines 40 through 42 of the match.go code file, we can connect the writing of the results, the closing of the channel, and the process- ing of results all together.

Listing 2.43 search/match.go: lines 38–42

38 // The channel blocks until a result is written to the channel. 39 // Once the channel is closed the for loop terminates. 40 for result := range results { 41 log.Printf("%s:\n%s\n\n", result.Field, result.Content) 42 }

The for range loop on line 40 of the match.go code file will block until a result is writ- ten to the channel. As each search goroutine writes its results to the channel (as you see on line 31 of the code file match.go), the for range loop wakes up and is given those results. The results are then immediately written to the log. It seems this for range loop is stuck in an endless loop, but it isn’t. Once the channel is closed on line 51 of the search.go code file, the for range loop is terminated and the Display function returns.

Before we look at the implementation of the RSS matcher, let’s review how the dif- ferent matchers are initialized when the program starts. To see this we need to look back at lines 07 through 10 of the default.go code file.

Listing 2.44 search/default.go: lines 06–10

06 // init registers the default matcher with the program. 07 func init() { 08 var matcher defaultMatcher 09 Register("default", matcher) 10 }

The default.go code file has a special function declared called init. You saw this function also declared in the main.go code file, and we talked about how all the init

Licensed to Mark Watson <nordickan@gmail.com>

**32** CHAPTER 2 ***Go quick-start***

functions in the program would be called before the main function begins. Let’s look at the imports again from the main.go code file.

Listing 2.45 main.go: lines 07–08

07 \_ "github.com/goinaction/code/chapter2/sample/matchers" 08 "github.com/goinaction/code/chapter2/sample/search"

The import to the search package on line 08 allows the compiler to find the init function in the default.go code file. Once the compiler sees the init function, it’s scheduled to be called prior to the main function being called.

The init function in the default.go code file is performing a special task. It’s creat- ing a value of the defaultMatcher type and passing that value to the Register func- tion that can be found in the search.go code file.

Listing 2.46 search/search.go: lines 59–67

59 // Register is called to register a matcher for use by the program. 60 func Register(feedType string, matcher Matcher) { 61 if \_, exists := matchers[feedType]; exists { 62 log.Fatalln(feedType, "Matcher already registered") 63 } 6465 log.Println("Register", feedType, "matcher") 66 matchers[feedType] = matcher 67 }

This function is responsible for adding the Matcher value to the map of registered matchers. All of this registration needs to happen before the main function gets called. Using init functions is a great way to accomplish this type of initialized registration.

***2.4 RSS matcher***

The last piece of code to review is the implementation of the RSS matcher. Everything we’ve reviewed up to now was to allow the implementation of different matcher types to run and search content within the program’s framework. The structure of the RSS matcher is similar to the structure of the default matcher. It’s the implementation of the interface method Search that’s different and in the end gives each matcher its uniqueness.

The RSS document in listing 2.47 shows you a sample of what we expect to receive when we use any link in the data feed that’s typed as an RSS feed.

Listing 2.47 Expected RSS feed document

<rss xmlns:npr="http://www.npr.org/rss/" xmlns:nprml="http://api

<channel>

<title>News</title> <link>...</link> <description>...</description>

Licensed to Mark Watson <nordickan@gmail.com>

**33 *RSS matcher***

<language>en</language> <copyright>Copyright 2014 NPR - For Personal Use <image>...</image> <item><title>

Putin Says He'll Respect Ukraine Vote But U.S. </title> <description>

The White House and State Department have called on the </description>

If you take any link from listing 2.47 and put it in a browser, you’ll be able to see a complete view of the expected RSS document. The implementation of the RSS matcher pulls down these RSS documents, searches the title and description fields for the search term, and sends the results over the results channel. Let’s start by looking at the first 12 lines of code for the rss.go code file.

Listing 2.48 matchers/rss.go: lines 01–12

01 package matchers 0203 import ( 04 "encoding/xml" 05 "errors" 06 "fmt" 07 "log" 08 "net/http" 09 "regexp" 1011 "github.com/goinaction/code/chapter2/sample/search" 12 )

As with every code file, we start on line 01 with the name of the package. This code file can be found in a folder called matchers, so the package name is matchers. Next we have six imports from the standard library and one import to the search package. Again, we have some packages from the standard library being imported from sub- folders within the standard library, such as xml and http. Just like with the json pack- age, the name of the last folder in the path represents the name of the package.

There are four struct types that are used to decode the RSS document, so we can use the document data in our program.

Listing 2.49 matchers/rss.go: lines 14–58

14 type ( 15 // item defines the fields associated with the item tag 16 // in the rss document. 17 item struct { 18 XMLName xml.Name `xml:"item"` 19 PubDate string `xml:"pubDate"` 20 Title string `xml:"title"` 21 Description string `xml:"description"`

Licensed to Mark Watson <nordickan@gmail.com>

**34** CHAPTER 2 ***Go quick-start***

22 Link string `xml:"link"` 23 GUID string `xml:"guid"` 24 GeoRssPoint string `xml:"georss:point"` 25 } 2627 // image defines the fields associated with the image tag 28 // in the rss document. 29 image struct { 30 XMLName xml.Name `xml:"image"` 31 URL string `xml:"url"` 32 Title string `xml:"title"` 33 Link string `xml:"link"` 34 } 3536 // channel defines the fields associated with the channel tag 37 // in the rss document. 38 channel struct { 39 XMLName xml.Name `xml:"channel"` 40 Title string `xml:"title"` 41 Description string `xml:"description"` 42 Link string `xml:"link"` 43 PubDate string `xml:"pubDate"` 44 LastBuildDate string `xml:"lastBuildDate"` 45 TTL string `xml:"ttl"` 46 Language string `xml:"language"` 47 ManagingEditor string `xml:"managingEditor"` 48 WebMaster string `xml:"webMaster"` 49 Image image `xml:"image"` 50 Item []item `xml:"item"` 51 } 5253 // rssDocument defines the fields associated with the rss document 54 rssDocument struct { 55 XMLName xml.Name `xml:"rss"` 56 Channel channel `xml:"channel"` 57 } 58 )

If you match these structures to the RSS document from any of the feed links, you’ll see how everything correlates. Decoding XML is identical to how we decoded JSON in the feed.go code file. Next we can look at the declaration of the rssMatcher type.

Listing 2.50 matchers/rss.go: lines 60–61

60 // rssMatcher implements the Matcher interface. 61 type rssMatcher struct{}

Again, this looks just like how we declared the defaultMatcher type. We use an empty struct since we don’t need to maintain any state; we just implement the Matcher inter- face. Next we have the implementation of the matcher init function.

Licensed to Mark Watson <nordickan@gmail.com>

**35 *RSS matcher***

Listing 2.51 matchers/rss.go: lines 63–67

63 // init registers the matcher with the program. 64 func init() { 65 var matcher rssMatcher 66 search.Register("rss", matcher) 67 }

Just like you saw with the default matcher, the init function registers a value of the rssMatcher type with the program for use. Let’s look at the import in the main.go code file once more.

Listing 2.52 main.go: lines 07–08

07 \_ "github.com/goinaction/code/chapter2/sample/matchers" 08 "github.com/goinaction/code/chapter2/sample/search"

The code in the main.go code file doesn’t directly use any identifiers from the matchers package. Yet we need the compiler to schedule the call to the init function in the rss.go code file. On line 07 we accomplish this by using the blank identifier as the alias name for the import. This allows the compiler to not produce an error for declaring the import and to locate the init function. With all of the imports, types, and initialization set, let’s look at the two remaining methods that support the imple- mentation of the Matcher interface.

Listing 2.53 matchers/rss.go: lines 114–140

114 // retrieve performs a HTTP Get request for the rss feed and decodes 115 func (m rssMatcher) retrieve(feed \*search.Feed)(\*rssDocument, error) { 116 if feed.URI == "" { 117 return nil, errors.New("No rss feed URI provided") 118 } 119 120 // Retrieve the rss feed document from the web. 121 resp, err := http.Get(feed.URI) 122 if err != nil { 123 return nil, err 124 } 125 126 // Close the response once we return from the function. 127 defer resp.Body.Close() 128 129 // Check the status code for a 200 so we know we have received a 130 // proper response. 131 if resp.StatusCode != 200 { 132 return nil, fmt.Errorf("HTTP Response Error %d\n",

resp.StatusCode) 133 } 134 135 // Decode the rss feed document into our struct type.

Licensed to Mark Watson <nordickan@gmail.com>

**36** CHAPTER 2 ***Go quick-start***

136 // We don't need to check for errors, the caller can do this. 137 var document rssDocument 138 err = xml.NewDecoder(resp.Body).Decode(&document) 139 return &document, err 140 }

The unexported method retrieve performs the logic for pulling the RSS document from the web for each individual feed link. On line 121 you can see the use of the Get method from the http package. In chapter 8 we’ll explore this package more, but for now Go makes it really easy to make web requests using the http package. When the Get method returns, we’ll get back a pointer to a value of type Response. After checking for errors, we need to schedule the call to the Close method, which we do on line 127.

On line 131 we check the StatusCode field of the Response value to verify we received a 200. Anything other than 200 must be handled as an error and we do just that. If the value isn’t 200, we then return a custom error using the Errorf function from the fmt package. The last three lines of code are similar to how we decoded the JSON data file. This time we use the xml package and call the same function named NewDecoder, which returns a pointer to a Decoder value. With the pointer, we call the Decode method passing the address of the local variable named document of type rssDocument. Then the address to the rssDocument type value and the error from the Decode method call are returned.

The final method to look at implements the Matcher interface.

Listing 2.54 matchers/rss.go: lines 69–112

69 // Search looks at the document for the specified search term. 70 func (m rssMatcher) Search(feed \*search.Feed, searchTerm string)

([]\*search.Result, error) { 71 var results []\*search.Result 72 73 log.Printf("Search Feed Type[%s] Site[%s] For Uri[%s]\n",

feed.Type, feed.Name, feed.URI) 74 75 // Retrieve the data to search. 76 document, err := m.retrieve(feed) 77 if err != nil { 78 return nil, err 79 } 80 81 for \_, channelItem := range document.Channel.Item { 82 // Check the title for the search term. 83 matched, err := regexp.MatchString(searchTerm,

channelItem.Title) 84 if err != nil { 85 return nil, err 86 } 87 88 // If we found a match save the result. 89 if matched { 90 results = append(results, &search.Result{ 91 Field: "Title", 92 Content: channelItem.Title,

Licensed to Mark Watson <nordickan@gmail.com>

**37 *RSS matcher***

93 }) 94 } 95 96 // Check the description for the search term. 97 matched, err = regexp.MatchString(searchTerm,

channelItem.Description) 98 if err != nil { 99 return nil, err 100 } 101 102 // If we found a match save the result. 103 if matched { 104 results = append(results, &search.Result{ 105 Field: "Description", 106 Content: channelItem.Description, 107 }) 108 } 109 } 110 111 return results, nil 112 }

We start on line 71 with the declaration of the results variable, which will be used to store and return any results that may be found.

Listing 2.55 matchers/rss.go: line 71

71 var results []\*search.Result

We use the keyword var and declare a nil slice of pointers to Result type values. The declaration of the Result type can be found again on line 08 of the match.go code file. Next on line 76 we make a web call using the retrieve method we just reviewed.

Listing 2.56 matchers/rss.go: lines 75–79

75 // Retrieve the data to search. 76 document, err := m.retrieve(feed) 77 if err != nil { 78 return nil, err 79 }

The call to the retrieve method returns a pointer to a value of type rssDocument and an error value. Then, as you’ve seen throughout the code, we check the error value for errors and return if there was an error. If no error exists, we then iterate through the results performing the match of the search term against the title and description of the retrieved RSS document.

Listing 2.57 matchers/rss.go: lines 81–86

81 for \_, channelItem := range document.Channel.Item { 82 // Check the title for the search term. 83 matched, err := regexp.MatchString(searchTerm,

channelItem.Title)

Licensed to Mark Watson <nordickan@gmail.com>

**38** CHAPTER 2 ***Go quick-start***

84 if err != nil { 85 return nil, err 86 }

Since the value of document.Channel.Item is a slice of item type values, we use a for range loop on line 81 to iterate through all the items. On line 83 we use the Match- String function from the regexp package to match the search term against the con- tent in the Title field of the channelItem value. Then we check for errors on line 84. If there are no errors, we move to lines 89 through 94 to check the results of the match.

Listing 2.58 matchers/rss.go: lines 88–94

88 // If we found a match save the result. 89 if matched { 90 results = append(results, &search.Result{ 91 Field: "Title", 92 Content: channelItem.Title, 93 }) 94 }

If the value of matched is true after the call to the MatchString method, we use the built-in function append to add the search results to the results slice. The built-in function append will grow the length and capacity of the slice as it needs to. You’ll learn more about the built-in function append in chapter 4. The first parameter to append is the value of the slice you want to append to, and the second parameter is the value you want to append. In our case, we use a struct literal to declare and initialize a value of type Result, and then we use the ampersand (&) operator to get the address of this new value, which is stored in the slice.

After the title is checked for matches, lines 97 through 108 perform the same logic again for the description field. Finally, on line 111, the method returns the results to the calling function.

***2.5 Summary***

▪ Every code file belongs to a package, and that package name should be the same as the folder the code file exists in.

▪ Go provides several ways to declare and initialize variables. If the value of a variable isn’t explicitly initialized, the compiler will initialize the variable to its zero value.

▪ Pointers are a way of sharing data across functions and goroutines.

▪ Concurrency and synchronization are accomplished by launching goroutines and using channels.

▪ Go provides built-in functions to support using Go’s internal data structures.

▪ The standard library contains many packages that will let you do some powerful things.

▪ Interfaces in Go allow you to write generic code and frameworks.

Licensed to Mark Watson <nordickan@gmail.com>

*Packaging and tooling*

***In this chapter***

▪ Understanding how Go code is organized

▪ Using the Go command

▪ Going farther with other Go developer tools

▪ Collaborating with other Go developers

In chapter 2 you got an overview of the syntax and language structure of Go. Now you’ll dive deeper into how code is organized into packages and how you interact with those packages. Packages are a critical concept in Go. The idea is to separate semantic units of functionality into different packages. When you do this, you enable code reuse and control the use of the data inside each package.

Before we get into the particulars, you should already be familiar with the com- mand prompt or system shell, and you should have Go installed according to the guidelines in the preface of this book. If you’re ready, let’s start by understanding what a package is and why it’s important in the Go ecosystem.

Licensed to Mark Watson <nordickan@gmail.com>

**39**

**40** CHAPTER 3 ***Packaging and tooling***

***3.1 Packages***

All Go programs are organized into groups of files called *packages*, so that code has the ability to be included into other projects as smaller reusable pieces. Let’s look at the packages that make up Go’s http functionality in the standard library:

net/http/

cgi/ cookiejar/

testdata/ fcgi/ httptest/ httputil/ pprof/ testdata/

These directories contain a series of related files with the .go extension, and provide clear separation of smaller units of code relating to the implementation of HTTP serv- ers, clients, and utilities to test and profile them. For example, the cookiejar package contains code related to storing and retrieving cookies from a web session. Each pack- age can be imported and used individually so that developers can import only the spe- cific functionality that they need. If you’re implementing an HTTP client, you only need to import the http package.

All .go files must declare the package that they belong to as the first line of the file excluding whitespace and comments. Packages are contained in a single directory. You may not have multiple packages in the same directory, nor may you split a pack- age across multiple directories. This means that all .go files in a single directory must declare the same package name.

***3.1.1 Package-naming conventions***

The convention for naming your package is to use the name of the directory contain- ing it. This has the benefit of making it clear what the package name is when you import it. If we continue with our example from the net/http package, all the files contained within the http directory are a part of the http package. When naming your packages and their directories, you should use short, concise, lowercase names, because they will be typed often while you’re developing. The packages under net/http are great examples of concise names such as cgi, httputil, and pprof.

Keep in mind that a unique name is not required, because you import the package using its full path. Your package name is used as the default name when your package is imported, but it can be overridden. This is beneficial when you need to import mul- tiple packages with the same name. We’ll discuss how this is done in section 3.2.

***3.1.2 Package main***

The package name main has special meaning in Go. It designates to the Go command that this package is intended to be compiled into a binary executable. All of the exe- cutable programs you build in Go must have a package called main.

Licensed to Mark Watson <nordickan@gmail.com>

**41 *Packages***

When the main package is encountered by the compiler, it must also find a function called main(); otherwise a binary executable won’t be created. The main() function is the entry point for the program, so without one, the program has no starting point. The name of the final binary will take the name of the directory the main package is declared in.

COMMANDS AND PACKAGES The Go documentation uses the term *command* frequently to refer to an executable program—like a command-line applica- tion. This can be confusing for new Go developers who are reading the docu- mentation. Remember that in Go, a command is any executable program, in contrast to a package, which generally means an importable semantic unit of functionality.

Go ahead and try it out. First start by creating a file called hello.go inside GOPATH/ src/hello/, and type the contents of listing 3.1 into it. This is the traditional Hello World! application again, but as you look at it, pay attention to the package declara- tion and import statements.

Listing 3.1 Traditional Hello World! application

01 package main 0203 import "fmt"

**The fmt package provides methods for performing formatted printing.** 0405 func main(){ 06 fmt.Println("Hello World!") 07 }GETTING PACKAGE DOCUMENTATION Don’t forget that you can get more details on a package by visiting http://golang.org/pkg/fmt/ or running godoc fmt from your terminal.

Once you’ve saved the file, you can run the command go build from within the GOPATH/src/hello/ directory. When it completes, you should see a binary file. On Unix, Linux, and Mac OS X this file will be named hello, whereas on Windows it will be called hello.exe. You can now run this application and see Hello World! printed to your console.

Had you named the package something other than main, like hello for instance, you’d have been telling the compiler this is just a package, not a command.

Listing 3.2 Invalid Go program with **main** function

01 package hello 0203 import "fmt" 0405 func main(){ 06 fmt.Println("Hello, World!") 07 }Licensed to Mark Watson <nordickan@gmail.com>

**42** CHAPTER 3 ***Packaging and tooling***

***3.2 Imports***

Now that we’ve looked at the organization of code into packages, we’ll take a look at how to import these individual packages so that you can access the code contained within them. The import statement tells the compiler where to look on disk to find the package you want to import. You import packages by using the keyword import, which tells the compiler that you want to reference the code contained within the package at that file location. If you need to import more than one package, the idio- matic way of doing so is to wrap the import statements in an import block, as demon- strated here.

Listing 3.3 Import statement blocksimport (

**The strings package provides many methods for searching,** "fmt" "strings" )**replacing, and transforming strings. You can get more details at http://golang.org/pkg/strings/ or by running “godoc strings” from your terminal.**

Packages are found on disk based on their relative path to the directories referenced by the Go environment. Packages in the standard library are found under where Go is installed on your computer. Packages that are created by you or other Go developers live inside the GOPATH, which is your own personal workspace for packages.

Let’s take a look at an example. If Go was installed under /usr/local/go and your GOPATH was set to /home/myproject:/home/mylibraries, the compiler would look for the net/http package in the following order:

/usr/local/go/src/pkg/net/http /home/myproject/src/net/http /home/mylibraries/src/net/http

**This is where the standard library source code is contained.**

The compiler will stop searching once it finds a package that satisfies the import state- ment. The important thing to remember is that the Go installation directory is the first place the compiler looks and then each directory listed in your GOPATH in the order that they’re listed.

If the compiler searches your GOPATH and never finds the package that you’ve ref- erenced, you’ll get an error when you try to run or build your program. You’ll see how to use the go get command to fix those problems later in this chapter.

***3.2.1 Remote imports***

There’s a huge trend toward sharing code via distributed version control systems (DVCS) such as sharing sites like GitHub, Launchpad, and Bitbucket. The Go tooling has built-in support for fetching source code from these sites and others. The import path can be used by the Go tooling to determine where the code you need fetched is on the network.

Licensed to Mark Watson <nordickan@gmail.com>

**43 *Imports***

For example:

import "github.com/spf13/viper"

When you try to build a program with this import path, the go build command will search the GOPATH for this package location on disk. The fact that it represents a URL to a repository on GitHub is irrelevant as far as the go build command is concerned. When an import path contains a URL, the Go tooling can be used to fetch the package from the DVCS and place the code inside the GOPATH at the location that matches the URL. This fetching is done using the go get command. go get will fetch any specified URL or can be used to fetch the dependencies a package is importing that are go- gettable. Since go get is recursive, it can walk down the source tree for a package and fetch all the dependencies it finds.

***3.2.2 Named imports***

What happens when you need to import multiple packages with the same name? For example, you could need a network/convert package for converting data that’s read from a network and a file/convert package for converting data read from text files. When this is the case, both of these packages can be imported by using *named imports*. This is performed by giving one of the packages a new name to the left of the import statement. As an example, let’s say you were already using the fmt package that comes as part of the standard library. Now you need to import a package named fmt that you had created as part of your own project. You can import your own fmt package by renam- ing the import, as demonstrated in the next listing.

Listing 3.4 Renaming imports

01 package main 0203 import ( 04 "fmt" 05 myfmt "mylib/fmt" 06 ) 0708 func main() { 09 fmt.Println("Standard Library") 10 myfmt.Println("mylib/fmt") 11 }

The Go compiler will fail the build and output an error whenever you import a pack- age that you don’t use. The Go team considers this a feature to eliminate code bloat from packages that are imported but not used. Although this feature is occasionally annoying, the Go team has put a great deal of effort into making decisions to prevent some of the problems that you encounter in other languages. You don’t want to have an unnecessarily large binary, filled with unused libraries, and they feel that if it’s worth the compiler telling you about, it’s worth failing the build. Anyone who has

Licensed to Mark Watson <nordickan@gmail.com>

**44** CHAPTER 3 ***Packaging and tooling***

compiled a large C program knows just how hard it can be to pinpoint the things that matter in a sea of compiler warnings.

Sometimes you may need to import a package that you don’t need to reference identifiers from. You’ll see why this might be useful in the next section. When this is the case, you can use the blank identifier \_ to rename an import.

BLANK IDENTIFIER The \_ (underscore character) is known as the *blank identi- fier* and has many uses within Go. It’s used when you want to throw away the assignment of a value, including the assignment of an import to its package name, or ignore return values from a function when you’re only interested in the others.

***3.3 init***

Each package has the ability to provide as many init functions as necessary to be invoked at the beginning of execution time. All the init functions that are discovered by the compiler are scheduled to be executed prior to the main function being exe- cuted. The init functions are great for setting up packages, initializing variables, or performing any other bootstrapping you may need prior to the program running.

An example of this is database drivers. They register themselves with the sql pack- age when their init function is executed at startup because the sql package can’t know about the drivers that exist when it’s compiled. Let’s look at an example of what an init function might do.

Listing 3.5 **init** function usage

01 package postgres 0203 import ( 04 "database/sql" **Creates new instance** 05 ) **of the postgres driver.** 06**We’ve intentionally** 07 func init() { 08 sql.Register("postgres", **left out its definition** new(PostgresDriver))

**to focus on init().** 09 }

This code lives inside your pretend database driver for the PostgreSQL database. When a program imports this package, the init function will be called, causing the database driver to be registered with Go’s sql package as an available driver.

In the program that we write using this new database driver, we’ll use the blank identifier to import the package so the new driver is included with the sql package. As stated earlier, you can’t import a package that you aren’t using, so renaming the import with the blank identifier allows the init function to be discovered and sched- uled to run without the compiler issuing an error about unused imports.

Now we can tell the sql.Open method to use this driver.

Licensed to Mark Watson <nordickan@gmail.com>

**45 *Using Go tools***

Listing 3.6 Blank identifier import aliasing

01 package main 0203 import ( 04 "database/sql" 05**Import driver anonymously to prevent compile error.** 06 \_ "github.com/goinaction/code/chapter3/dbdriver/postgres" 07 ) 0809 func main() { **We call the Open method provided by the sql package; this works because** 10 sql.Open("postgres", "mydb") **the driver registered itself with the sql** 11 }

**package in its init function.**

***3.4 Using Go tools***

We’ve been working with the go tool for a few chapters now, but we haven’t explored all it can do. Let’s dive a little deeper into this diminutively named powerhouse and explore more of its capabilities. From a shell prompt, type the go command with no arguments:

$ go

As you can see in figure 3.1, there are a lot of features buried in the go tooling.

Figure 3.1 Output of **go** command help text

Licensed to Mark Watson <nordickan@gmail.com>

**46** CHAPTER 3 ***Packaging and tooling***

Looking through the list, you can see that there really is a compiler in there; it’s used by the build command. The build and clean commands do exactly what you’d expect them to do. Try them now using the source code from listing 3.2:

go build hello.go

You might not want that file hanging around when it’s time to check your code into source control. To get rid of it, use the clean command:

go clean hello.go

After calling clean, the executable program is gone. Let’s take a closer look at some of the features of the go tool, and ways that you can save time when you’re using it. For the next examples, we’ll use the sample code in the following listing.

Listing 3.7 Working with the **io** package

01 package main 0203 import ( 04 "fmt" 05 "io/ioutil" 06 "os" 0708 "github.com/goinaction/code/chapter3/words" 09 ) 1011 // main is the entry point for the application. 12 func main() { 13 filename := os.Args[1] 1415 contents, err := ioutil.ReadFile(filename) 16 if err != nil { 17 fmt.Println(err) 18 return 19 } 2021 text := string(contents) 2223 count := words.CountWords(text) 24 fmt.Printf("There are %d words in your text. \n", count) 25 }

If you’ve downloaded the source code for the book, this package should be at GOPATH/src/github.com/goinaction/code/chapter3/words. Make sure you have it there to follow along.

Most of the commands that are part of the Go tooling take a package specifier as an argument. Look closer at the commands we’ve just used, and you’ll see one of the shortcuts built into the tooling. You can omit the filename of the source code file that you want to build, and the go tool will default to *the current package*:

go build

Licensed to Mark Watson <nordickan@gmail.com>

**47 *Going farther with Go developer tools***

Building a package is a common practice, and the package can also be specified directly:

go build github.com/goinaction/code/chapter3/wordcount

You can also specify wildcards in your package specifiers. Three periods in your pack- age specifier indicate a pattern matching any string. For example, the following com- mand will build every package under the chapter3 directory:

go build github.com/goinaction/code/chapter3/...

Instead of a package specifier, you can also use a path shortcut as an argument to most of the Go commands. For example, you could achieve the same effect with these two commands:

go build wordcount.go

go build .

To execute this program, you need to run the wordcount or wordcount.exe program that was created after the build. But there’s a different command that can perform both operations in a single call:

go run wordcount.go

The go run command both builds and executes the program contained in wordcount .go, which saves a lot on typing.

You’ll use the go build and go run commands the most when you’re developing. Let’s take a look at a few of the other available commands and see what they can do.

***3.5 Going farther with Go developer tools***

You’ve seen how to compile and run your Go programs using the convenient go tool. But that handy little developer tool has a lot of other tricks hidden inside.

***3.5.1 go vet***

It won’t write code for you, but once you’ve written some code, the vet command will check your code for common errors. Let’s look at the types of errors vet can catch:

▪ Bad parameters in Printf-style function calls

▪ Method signature errors for common method definitions

▪ Bad struct tags

▪ Unkeyed composite literals

Let’s look at a mistake many new Go developers make. The fmt.Printf function is a great way to produce formatted output, but the function requires you to remember all the different format specifiers. The following listing is an example.

Licensed to Mark Watson <nordickan@gmail.com>

**48** CHAPTER 3 ***Packaging and tooling***

Listing 3.8 Working with **go vet**

01 package main 0203 import "fmt" 0405 func main() { 06 fmt.Printf("The quick brown fox jumped over lazy dogs", 3.14) 07 }

This program inserts the floating-point number 3.14, but there’s no placeholder in the formatted string. If you run go vet against this source code, you get the following message:

go vet main.go

main.go:6: no formatting directive in Printf call

The go vet tool won’t keep you from making huge errors in logic, or from creating buggy code. However, as you can see from the last example, it does catch some com- mon errors quite nicely. It’s a great idea to get in the habit of running go vet on your code base before you commit it to a source repository.

***3.5.2 Go format***

The fmt command is a favorite in the Go community. Instead of arguing about where curly braces should go, or whether to use tabs or spaces when you indent, the fmt tool makes these decisions moot by applying a predetermined layout to Go source code. To invoke this code formatter, type go fmt followed by a file or package specification. The fmt command will automatically format the source code files you specify and save them. Here’s a before-and-after snapshot of a few lines of code run through go fmt:

if err != nil { return err }

After running go fmt on this code, you’ll get the following:

if err != nil {

return err }Many Go developers configure their development environment to perform a go fmt on save or before committing to a code repository. Do yourself a favor and configure this right now.

***3.5.3 Go documentation***

There’s another tool that will make your Go development process easier. Go has two ways to deliver documentation to developers. If you’re working at a command prompt, you can use the go doc command to print documentation directly to your ter- minal session. You can view a quick reference for a command or package without leav- ing your terminal. But if a browsable interface is more your speed, you can use the

Licensed to Mark Watson <nordickan@gmail.com>

**49 *Going farther with Go developer tools***

godoc program to start a web server with a clickable index of Go packages. The godoc web server gives you a fully navigable web version of the documentation for all the Go source code installed in your system.

GETTING DOCUMENTATION AT THE COMMAND LINE If you’re the kind of developer who has a text editor open and a terminal session open right next to it (or a text editor open in your terminal session), then go doc is going to be your tool of choice. The first time you need to read a Unix tar file from your Go application, you’ll be happy that you can access the documentation for the archive/tar package by simply typing this:

go doc tar

Running this command produces the following output, directly to the terminal:

PACKAGE DOCUMENTATION

package tar // import "archive/tar"

Package tar implements access to tar archives. It aims to cover most of the variations, including those produced by GNU and BSD tars.

References:

http://www.freebsd.org/cgi/man.cgi?query=tar&sektion=5 http://www.gnu.org/software/tar/manual/html\_node/Standard.html http://pubs.opengroup.org/onlinepubs/9699919799/utilities/pax.html var ErrWriteTooLong = errors.New("archive/tar: write too long") ... var ErrHeader = errors.New("archive/tar: invalid tar header") func FileInfoHeader(fi os.FileInfo, link string) (\*Header, error) func NewReader(r io.Reader) \*Reader func NewWriter(w io.Writer) \*Writer type Header struct { ... } type Reader struct { ... } type Writer struct { ... }

You can skim through the documentation and find the information you need without breaking your workflow.

BROWSING THE DOCUMENTATION The Go documentation is also available in a browsable format. Sometimes it’s easier to get the whole picture of a package or function when you can click around and see all the related details. For those cases, you’ll want to use godoc as a web server. If you pre- fer to get your documentation from a web browser in a clickable format, then this will be your favorite way to get to the documentation.

To start your own documentation server, type the following command into a termi- nal session:

godoc -http=:6060

This command instructs godoc to start a web server on port 6060. If you open your web browser and navigate to http://localhost:6060, you’ll see a web page with documenta- tion for both the Go standard libraries and any Go source that lives in your GOPATH.

Licensed to Mark Watson <nordickan@gmail.com>

**50** CHAPTER 3 ***Packaging and tooling***

If the documentation in figure 3.2 looks familiar to you, it’s because a slightly modi- fied version of godoc is serving up the documentation for the Go website. To navigate to the documentation for a specific package, just click the Packages link at the top of the page. The best part of Go’s documentation tool is that it works for your code, too. If you follow simple conventions while writing your code, it will automatically include your comments in the Go documentation generated by godoc.

To be included in the godoc generated documentation, your code needs to be doc- umented by adding comments that follow a specific convention. We won’t go through the whole convention in this chapter, but we’ll hit the highlights.

Start by adding comments directly above the identifiers you want to document. This works for packages, functions, types, and global variables. Comments can be started using either two slashes, or using the slash-asterisk style.

// Retrieve connects to the configuration repository and gathers // various connection settings, usernames, passwords. It returns a // config struct on success, or an error. func Retrieve() (config, error) {

// ... omitted }

Figure 3.2 Local Go documentation

Licensed to Mark Watson <nordickan@gmail.com>

**51 *Collaborating with other Go developers***

In this example, we show the idiomatic way to document a function in Go. The docu- mentation for the function immediately precedes the function and is written in com- plete sentences. If you want to add a large body of text to document your package, include a file called doc.go that declares the same package as your project, and put your package introduction as a comment before the package declaration:

/\*

Package usb provides types and functions for working with USB devices. To connect to a USB device start by creating a new USB connection with NewConnection ... \*/package usb

This package documentation will be shown before any type or function documenta- tion is displayed for your package. It also demonstrates using the slash-asterisk type of comment. You can read more about creating good documentation for your code by searching for *golang documentation* in Google.

***3.6 Collaborating with other Go developers***

Modern developers don’t code in a vacuum, and the Go tooling recognizes and embraces this fact. The concept of packages extends beyond your local development environment, thanks to the go tool. Let’s look at a few conventions to follow in order to be a good citizen in a distributed development environment.

***3.6.1 Creating repositories for sharing***

Once you start cranking out awesome Go code, you’re probably going to want to share that code with the rest of the Go community. It’s really easy as long as you follow a few simple steps.

PACKAGE SHOULD LIVE AT THE ROOT OF THE REPOSITORY When you’re using go get, you specify the full path to the package that should be imported. This means that when you create a repository that you intend to share, the package name should be the repository name, and the package’s source should be in the root of the repository’s directory structure.

A common mistake that new Go developers make is to create a code or src direc- tory in their public repository. Doing so will make the package’s public import longer. Instead, just put the package source files at the root of the public repository.

PACKAGES CAN BE SMALL It’s common in Go to see packages that are relatively small by the standards of other programming languages. Don’t be afraid to make a package that has a small API or performs only a single task. That’s normal and expected.

RUN GO FMT ON THE CODE Just like any other open source repository, people will look at your code to gauge the quality of it before they try it out. You need to be running go fmt before checking

Licensed to Mark Watson <nordickan@gmail.com>

**52** CHAPTER 3 ***Packaging and tooling***

anything in. It makes your code readable and puts everyone on the same page when reading source code.

DOCUMENT THE CODE Go developers use godoc to read documentation, and http://godoc.org to read docu- mentation for open source packages. If you’ve followed go doc best practices in documenting your code, your packages will appear well documented when viewed locally or online, and people will find it easier to use.

***3.7 Dependency management***

The community has been hard at work since the release of Go 1.0 to provide Go tool- ing that makes life easier for developers. Many of these tools focus on helping with dependency management. The most popular tools today are *godep* by Keith Rarick, *vendor* by Daniel Theophanes, and a tool by Gustavo Niemeyer called *gopkg.in*, which helps package authors publish different versions of their packages.

As a call to action, with version 1.5 the Go language team started to experiment with new build options and features to provide better internal tooling support for dependency management. While we wait today to see where these experiments lead, there are existing tools that provide the ability to manage, build, and test Go code in a reproducible way.

***3.7.1 Vendoring dependencies***

Community tools such as godep and vendor have solved the dependency problem by using a technique called *vendoring* and import path rewriting. The idea is to copy all the dependencies into a directory inside the project repo, and then rewrite any import paths that reference those dependencies by providing the location inside the project itself.

Listing 3.9 Project using godep

$GOPATH/src/github.com/ardanstudios/myproject

|-- Godeps | |-- Godeps.json | |-- Readme | |-- \_workspace | |-- src | |-- bitbucket.org | |-- ww | | |-- goautoneg | | |-- Makefile | | |-- README.txt | | |-- autoneg.go | | |-- autoneg\_test.go | |-- github.com | |-- beorn7 | |-- perks | |-- README.md | |-- quantile

Licensed to Mark Watson <nordickan@gmail.com>

**53 *Dependency management***

| |-- bench\_test.go | |-- example\_test.go | |-- exampledata.txt | |-- stream.go ||-- examples |-- model |-- README.md |-- main.go

Listing 3.9 shows a typical source tree when using godep to vendor the dependencies for a project. You can see how godep created a directory called Godeps. The source code for the dependencies that the tooling vendored is located inside another set of directories called \_workspace/src.

Next, if you look at the import statements that are declared inside of main.go for these dependencies, you’ll see that some things needed to change.

Listing 3.10 Before vendoring

01 package main 0203 import ( 04 "bitbucket.org/ww/goautoneg" 05 "github.com/beorn7/perks" 06 )

Listing 3.11 After vendoring

01 package main 0203 import ( 04 "github.ardanstudios.com/myproject/Godeps/\_workspace/src/

bitbucket.org/ww/goautoneg" 05 "github.ardanstudios.com/myproject/Godeps/\_workspace/src/

github.com/beorn7/perks" 06 )

Before the dependencies were vendored, the import statements used the canonical path for the package. The code was physically located on disk within the scope of GOPATH. After vendoring, import path rewriting became necessary to reference the packages, which are now physically located on disk inside the project itself. You can see these imports are very large and tedious to use.

With vendoring, you have the ability to create reproducible builds, since all the source code required to build the binary is housed inside the single project repo. One other benefit of vendoring and import path rewriting is that the project repo is still go-gettable. When go get is called against the project repo, the tooling can find each package and store the package exactly where it needs to be inside the project itself.

Licensed to Mark Watson <nordickan@gmail.com>

**54** CHAPTER 3 ***Packaging and tooling***

***3.7.2 Introducing gb***

Gb is a whole new class of build tool being developed by members of the Go commu- nity. Gb takes a different approach to solving the reproducible-build problem, which starts with the understanding that wrapping the Go tooling is not an option.

The philosophy behind gb stems from the idea that Go doesn’t have reproducible builds because of the import statement. The import statement drives go get, but import doesn’t contain sufficient information to identify which revision of a package should be fetched any time go get is called. The possibility that go get can fetch a dif- ferent version of code for any given package at any time makes supporting the Go tooling in any reproducible solution complicated and tedious at best. You saw some of this tediousness with the large import paths when using godep.

This understanding resulted in the creation of the gb build tool. Gb doesn’t wrap the Go tooling, nor does it use GOPATH. Gb replaces the Go tooling workspace meta- phor with a project-based approach. This has natively allowed vendoring without the need for rewriting import paths, which is mandated by go get and a GOPATH workspace.

Let’s look at how the last project could be converted into a gb project.

Listing 3.12 Example of a gb project

/home/bill/devel/myproject ($PROJECT) |-- src | |-- cmd | | |-- myproject | | | |-- main.go | |-- examples | |-- model | |-- README.md |-- vendor

|-- src

|-- bitbucket.org | |-- ww | |-- goautoneg | |-- Makefile | |-- README.txt | |-- autoneg.go | |-- autoneg\_test.go |-- github.com |-- beorn7

|-- perks |-- README.md |-- quantile |-- bench\_test.go |-- example\_test.go |-- exampledata.txt |-- stream.go

A gb project is simply a directory on disk that contains a subdirectory named src/. The symbol $PROJECT refers to the root directory on disk where the src/ directory is located and is only used as a shortcut for describing the location on disk for the project.

Licensed to Mark Watson <nordickan@gmail.com>

**55 *Dependency management***

$PROJECT is *not* an environmental variable that needs to be set. In fact, gb requires no environmental variables to be set at all.

Gb projects differentiate between the code you write and the code your code depends on. The code your code depends on is called *vendored code*. A gb project makes a clear distinction between your code and vendored code.

Listing 3.13 The location for the code you write for the project

$PROJECT/src/

Listing 3.14 The location of vendored code

$PROJECT/vendor/src/

One of the best things about gb is that there’s no need for import path rewriting. Look at the import statements that are declared inside of main.go—nothing needs to change to reference the vendored dependencies.

Listing 3.15 Import paths for gb projects

01 package main 0203 import ( 04 "bitbucket.org/ww/goautoneg" 05 "github.com/beorn7/perks" 06 )

The gb tool will look inside the $PROJECT/vendor/src/ directory for these imports if they can’t be located inside the $PROJECT/src/ directory first. The entire source code for the project is located within a single repo and directory on disk, split between the src/ and vendor/src/ subdirectories. This, in conjunction with no need to rewrite import paths and the freedom to place your project anywhere you wish on disk, makes gb a popular tool in the community to develop projects that require reproduc- ible builds.

One thing to note: a gb project is not compatible with the Go tooling, including go get. Since there’s no need for GOPATH, and the Go tooling doesn’t understand the structure of a gb project, it can’t be used to build, test, or get. Building and testing a gb project requires navigating to the $PROJECT directory and using the gb tool.

Listing 3.16 Building a gb project

gb build all

Many of the same features that are supported by the Go tooling are supported in gb. Gb also has a plugin system to allow the community to extend support. One such plugin is called vendor, which provides conveniences to manage the dependencies in

Licensed to Mark Watson <nordickan@gmail.com>

**56** CHAPTER 3 ***Packaging and tooling***

the $PROJECT/vendor/src/ directory, something the Go tooling does not have today. To learn more about gb, check out the website: getgb.io.

***3.8 Summary***

▪ Packages are the basic unit of code organization in Go.

▪ Your GOPATH determines on disk where Go source code is saved, compiled, and installed.

▪ You can set your GOPATH for each different project, keeping all of your source and dependencies separate.

▪ The go tool is your best friend when working from the command line.

▪ You can use packages created by other people by using go get to fetch and install them in your GOPATH.

▪ It’s easy to create packages for others to use if you host them on a public source code repository and follow a few simple rules.

▪ Go was designed with code sharing as a central driving feature of the language.

▪ It’s recommended that you use vendoring to manage dependencies.

▪ There are several community-developed tools for dependency management such as godep, vendor, and gb.

Licensed to Mark Watson <nordickan@gmail.com>

*Arrays, slices, and maps*

***In this chapter***

▪ Array internals and fundamentals

▪ Managing collections of data with slices

▪ Working with key/value pairs using maps

It’s difficult to write programs that don’t need to store and read collections of data. If you use databases or files, or access the web, you need a way to handle the data you receive and send. Go has three different data structures that allow you to man- age collections of data: arrays, slices, and maps. These data structures are baked into the language and used throughout the standard library. Once you learn how these data structures work, programming in Go will become fun, fast, and flexible.

***4.1 Array internals and fundamentals***

It makes sense to start with arrays because they form the base data structure for both slices and maps. Understanding how arrays work will help you appreciate the elegance and power that slices and maps provide.

Licensed to Mark Watson <nordickan@gmail.com>

**57**

**58** CHAPTER 4 ***Arrays, slices, and maps***

***4.1.1 Internals***

An array in Go is a fixed-length data type that contains a contiguous block of elements of the same type. This could be a built-in type such as integers and strings, or it can be a struct type.

In figure 4.1 you can see the representation of an array. The elements of the array are marked as a grey box and are connected in series to each other. Each element con- tains the same type, in this case an integer, and can be accessed through a unique index position.

[ 0 ] [ 1 ] [ 2 ] [ 3 ] [ 4 ]

0 Integer Arrays are valuable data structures because the memory is allocated sequentially. Hav- ing memory in a contiguous form can help to keep the memory you use stay loaded within CPU caches longer. Using index arithmetic, you can iterate through all the ele- ments of an array quickly. The type information for the array provides the distance in memory you have to move to find each element. Since each element is of the same type and follows each other sequentially, moving through the array is consistent and fast.

***4.1.2 Declaring and initializing***

An array is declared by specifying the type of data to be stored and the total number of elements required, also known as the array’s length.

// Declare an integer array of five elements. var array [5]int

Once an array is declared, neither the type of data being stored nor its length can be changed. If you need more elements, you need to create a new array with the length needed and then copy the values from one array to the other.

When variables in Go are declared, they’re always initialized to their zero value for their respective type, and arrays are no different. When an array is initialized, each individual element that belongs to the array is initialized to its zero value. In figure 4.2, you can see an array of integers with each element in the array initialized to 0, the zero value for integers.

20 Integer 30

Integer 40

Integer 50

Integer Figure 4.1

Array internals

Listing 4.1 Declaring an array set to its zero value

Licensed to Mark Watson <nordickan@gmail.com>

**59 *Array internals and fundamentals***

[ 0 ]

[ 1 ]

[ 2 ]

[ 3 ]

[ 4 ]

0

0

0

0

0 Integer

Integer

Integer

Integer

Integer

Figure 4.2 Values of the array after the declaration of the array variable

A fast and easy way to create and initialize arrays is to use an array literal. Array liter- als allow you to declare the number of elements you need and specify values for those elements.

Listing 4.2 Declaring an array using an array literal

// Declare an integer array of five elements. // Initialize each element with a specific value. array := [5]int{10, 20, 30, 40, 50}

If the length is given as ..., Go will identify the length of the array based on the num- ber of elements that are initialized.

Listing 4.3 Declaring an array with Go calculating size

// Declare an integer array. // Initialize each element with a specific value. // Capacity is determined based on the number of values initialized. array := [...]int{10, 20, 30, 40, 50}

If you know the length of the array you need, but are only ready to initialize specific elements, you can use this syntax.

Listing 4.4 Declaring an array initializing specific elements

// Declare an integer array of five elements. // Initialize index 1 and 2 with specific values. // The rest of the elements contain their zero value. array := [5]int{1: 10, 2: 20}

The values for the array declared in listing 4.4 will look like figure 4.3 after the array is declared and initialized.

[ 0 ] [ 1 ] [ 2 ] [ 3 ] [ 4 ]

0

10

20

0

0 Integer Integer Integer Integer Integer

Figure 4.3 Values of the array after the declaration of the array variable

Licensed to Mark Watson <nordickan@gmail.com>