O'REILLY®

Go Cookbook

Expert Solutions for Commonly Needed Go Tasks

Sau Sheong Chang

O'REILLY®

Go Cookbook

Go is an increasingly popular language for programming everything from web applications to distributed network services. While Go is relatively easy and familiar for most programmers coming from the C/Java tradition, there are enough differences for developers to wonder, *How can I do this in Go?*

This practical guide provides recipes to help you unravel common problems and perform useful tasks when working with Go. Each recipe includes self-contained code solutions that you can freely use, along with a discussion of how and why they work. Programmers new to Go can quickly ramp up their knowledge while accomplishing useful tasks, and experienced Go developers can save time by cutting and pasting proven code directly into their applications.

Recipes in this guide will help you:

- **Create a module**
- **Call code from another module**
- **Return and handle an error**
- **Convert strings to numbers (or convert numbers to strings)**
- **Modify multiple characters in a string**
- **Create substrings from a string**
- **Capture string input**
- **And so much more**

"Developers new to Go often want to quickly implement a common task but don't know the best way to do so. Sau Sheong Chang's *Go Cookbook* provides answers to these questions. It's a good resource for discovering Go's standard library."

—Jon Bodner
Author of *Learning Go*
and Staff Engineer, Datadog

Sau Sheong Chang, a 28-year veteran of the software development industry, has been involved in building software products in many industries using various technologies. He's been an active member of the software development communities for Java and Ruby as well as for Go. He runs meetups and gives talks in conferences all around the world.

OTHER PROGRAMMING LANGUAGES

US $79.99 CAN $99.99
ISBN: 978-1-098-12211-9

Twitter: @oreillymedia
linkedin.com/company/oreilly-media
youtube.com/oreillymedia

Praise for *Go Cookbook*

Developers new to Go often want to quickly implement a common task but don't know the best way to do so. Sau Sheong Chang's *Go Cookbook* provides answers to these questions. It's a good resource for discovering Go's standard library.

—Jon Bodner, Staff Engineer, Datadog and author of Learning Go

Go Cookbook is an indispensable companion for seasoned programmers looking to unleash the full potential of Go. Its practical recipes, covering everything from error handling and concurrency to networking and testing, and much more, make it a go-to resource for solving practical challenges. A must-have for any Go developer.

—Alex Chen, CTO, FMZ Quant

This book is a great asset to any Go developers, it covers many daily essential recipes that any go developer would face including module management, logging, string manipulations, file IO, structs and interfaces, http server creation, unit testing and benchmarking.

—Jason Wong Ho Chi, Senior Solution Engineer on APM of AppDynamics, Go hobbyist, Elasticsearch specialist, and trainer

The tips provided throughout the book are quite practical for production use, and the organization of chapters into problem-solution subsections along with the code snippets makes topics like testing and benchmarking easy to comprehend. A must-read for entry-level software engineers.

—Vaibhav Jain, Senior Software Engineer, Ninja Van (and currently a Master's student, Georgia Institute of Technology)

Sau Sheong's *Go Cookbook* takes the developer on a brief yet rich tour through some of Go's more intricate areas. Actionable examples coupled with clear explanations quickly empower the developer to adopt alternative approaches, libraries and techniques. The book is relevant for a wide range of backgrounds from beginner to experts alike wanting to brush up on recent language features critically including coverage and fuzzing.

—Daniel J Blueman, Principal Software Engineer, Numascale AS

Go Cookbook by Sau Sheong Chang is a great resource for those who want to learn Go quickly. It covers essential topics in a straightforward manner. I personally found it to be very useful as a reference guide.

—Carlos Alexandre Queiroz, Global Head of Data Science Engineering at a global bank

Go Cookbook

Expert Solutions for Commonly Needed Go Tasks

Sau Sheong Chang

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Go Cookbook
by Sau Sheong Chang

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editors: Zan McQuade; Brian Guerin
Development Editor: Shira Evans
Production Editor: Elizabeth Faerm
Copyeditor: Kim Cofer
Proofreader: Piper Editorial Consulting, LLC
Indexer: Potomac Indexing, LLC
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

September 2023: First Edition

Revision History for the First Edition
2023-09-13: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098122119* for release details.

978-1-098-12211-9

[LSI]

Table of Contents

Preface

Go has been around for more than 10 years. It was publicly announced in 2009, and version 1.0 was released in March 2012. Since 2013 it has gained a steady rise in popularity and is frequently listed among the top 10 most popular programming languages in use today. In the past 10 years there have been plenty of books written about Go, including *Go Web Programming*, which I wrote in 2015. Most of what needs to be written about Go has already been written; however, the language continues to evolve, and there are new generations of would-be Go programmers coming on board.

This book came about because of a podcast interview. In September 2021, in the middle of the pandemic, Natalie Pistunovich hosted a "Go Time" podcast interview, titled "Books that Teach Go," with my friend, Bill Kennedy, and me regarding our Go books. I spoke about *Go Web Programming* and my new blog site, *Go Recipes*, which teaches readers how to do the basic stuff with Go. I wanted to provide a steady stream of know-how to serve as a guide for both would-be and experienced Go programmers.

After the podcast, Natalie mentioned that, coincidentally, O'Reilly was looking for someone to write a Go cookbook. Since I already had been writing Go recipes, I thought it was too much of a fateful encounter to ignore. Natalie put me in contact with O'Reilly, and the rest became history (and is now part of the Preface)!

This cookbook, like many others, is not about teaching new or specific topics but instead explains the basics of common tasks. It covers as much ground as possible on what programmers are most likely to use. The coverage is wide, rather than comprehensive. Each recipe is, more or less, standalone; although at times I cite other recipes, it is not necessary to reference them. You may find some recipes either boring or simple, but there are plenty to choose from!

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/sausheong/gocookbook*.

If you have a technical question or a problem using the code examples, please send email to *support@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Go Cookbook* by Sau Sheong Chang (O'Reilly). Copyright 2023 Sau Sheong Chang, 978-1-098-12211-9."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-889-8969 (in the United States or Canada)
707-829-7019 (international or local)
707-829-0104 (fax)
support@oreilly.com
https://www.oreilly.com/about/contact.html

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/go-cookbook*.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*.

Follow us on Twitter: *https://twitter.com/oreillymedia*.

Watch us on YouTube: *https://youtube.com/oreillymedia*.

Acknowledgments

I would like to thank my wife, Angela Lim, and my son, Chang Kai Wen, for bearing with me while I write "just one more book," for supporting me through this journey, and for their patience over the long weekends and late nights. It can be tough being the family of a writer, but they took it on admirably well (I think).

I want to acknowledge Natalie Pistunovich who not only introduced me to O'Reilly but also helped me review the early chapters of this book. Also, Jon Bodner and Jess Males, both of whom helped me tremendously with their reviews, encouragement, and great suggestions.

I also want to thank my current and former colleagues from SP Group, Temasek, and GovTech Singapore for their support and encouragement. The list is too long, but you know who you are and I thank you for your constant support!

A writer is nothing without his readers. I appreciate and want to thank all the readers of my blog articles and those who followed my writings on Go as well as various other technologies on my blog site. Your continued support is a great motivator, and I'm very grateful that you find it worthwhile to spend time reading them. Thank you!

Finally, I want to thank my late father, Chang Yoon Sang, who passed away last November. He always supported me unconditionally and enthusiastically in everything I do. He was especially excited that I followed in his footsteps in writing books (he was a traditional Chinese medicine physician and nutritionist, and wrote on both topics, in Chinese). Even though I know he struggled to read my books (he's not a technical person), I know he was always proud of me, asking me for copies of all my books, including the translations, and wanting to build a shelf full of books his son wrote. My biggest regret is that he never had the chance to add this book to his collection.

This book is dedicated to him.

CHAPTER 1

Getting Started Recipes

1.0 Introduction

Let's start easy first. In this chapter, you'll go through the essential recipes to begin coding in Go, installing Go, and then writing some simple Go code. You will go through the fundamentals, from using external libraries and handling errors to simple testing and logging events. Some of these recipes are intentionally concise—more detail about them in later chapters. If you're already familiar with the basics of Go, you can skip this chapter altogether.

1.1 Installing Go

Problem

You want to install Go and prepare the environment for Go development.

Solution

Go to the Go website and download the latest, greatest version of Go. Then follow the installation instructions to install Go.

Discussion

First, you need to go to the Go download site. You can choose the correct version according to your operating system and hardware and download the right package.

There are a couple of ways to install Go—you can either install the prebuilt binaries for your platform or compile it from the source. Most of the time, there is no need to compile it from the source unless you can't find the appropriate prebuilt binaries for your operating system. Even then, you can usually use your operating system's package manager to install Go instead.

MacOS

Open the downloaded package file and follow the prompts to install. Go will be installed at */usr/local/go*, and your `PATH` environment variable should also have `/usr/local/go/bin` added to it.

You can also choose to install using Homebrew, which is usually a version or two behind. To do this, run this from the command line:

```
$ brew update && brew install golang
```

You can set up the `PATH` later as you like.

Linux

Extract the downloaded archive file into */usr/local*, which should create a `go` directory. For example, run this from the command line (replace the Go version as necessary):

```
$ rm -rf /usr/local/go && tar -C /usr/local -xzf go1.20.1.linux-amd64.tar.gz
```

You can add */usr/local/go/bin* into your `PATH` as needed by adding this line to your *$HOME/.profile*:

```
export PATH=$PATH:/usr/local/go/bin
```

The changes will be made the next time you log in, or if you want it to take effect immediately, run `source` on your profile file:

```
$ source $HOME/.profile
```

Windows

Open the downloaded MSI installer file and follow the prompts to install Go. By default, Go will be installed to `Program Files` or `Program Files (x86)`, but you can always change this.

Build from source

You shouldn't build from source unless you cannot find the prebuilt binaries for your operating system. However, if you need to, extract the source files to an appropriate directory first, then run these commands from the command line:

```
$ cd src
$ ./all.bash
```

If you're building in Windows, use `all.bat` instead. The assumption here is that the compilers are already in place. If not, and if you need a deeper dive into installing Go, go to the Installing Go site for details.

If all goes well, you should see something like this:

```
ALL TESTS PASSED

---
Installed Go for linux/amd64 in /home/you/go.
Installed commands in /home/you/go/bin.
*** You need to add /home/you/go/bin to your $PATH. ***
```

To check if you have installed Go, you can run the Go tool with the `version` option to see the installed version:

```
% go version
```

If you have correctly installed Go, you should see something like this:

```
go version go1.20.1 darwin/amd64
```

1.2 Playing Around with Go

Problem

You want to write and execute Go code without downloading and installing Go.

Solution

Use the Go Playground to run your Go code.

Discussion

The Go Playground runs on Google's servers, and you can run your program inside a sandbox. Go to the Go Playground URL on your browser. This online environment lets you play with Go code, running the latest Go version (you can switch to a different version). The same web page returns output but only supports standard output and standard error (see Figure 1-1).

In a pinch, the Go Playground is a good option to test some Go code. You can also use the Go Playground to share executable code directly.

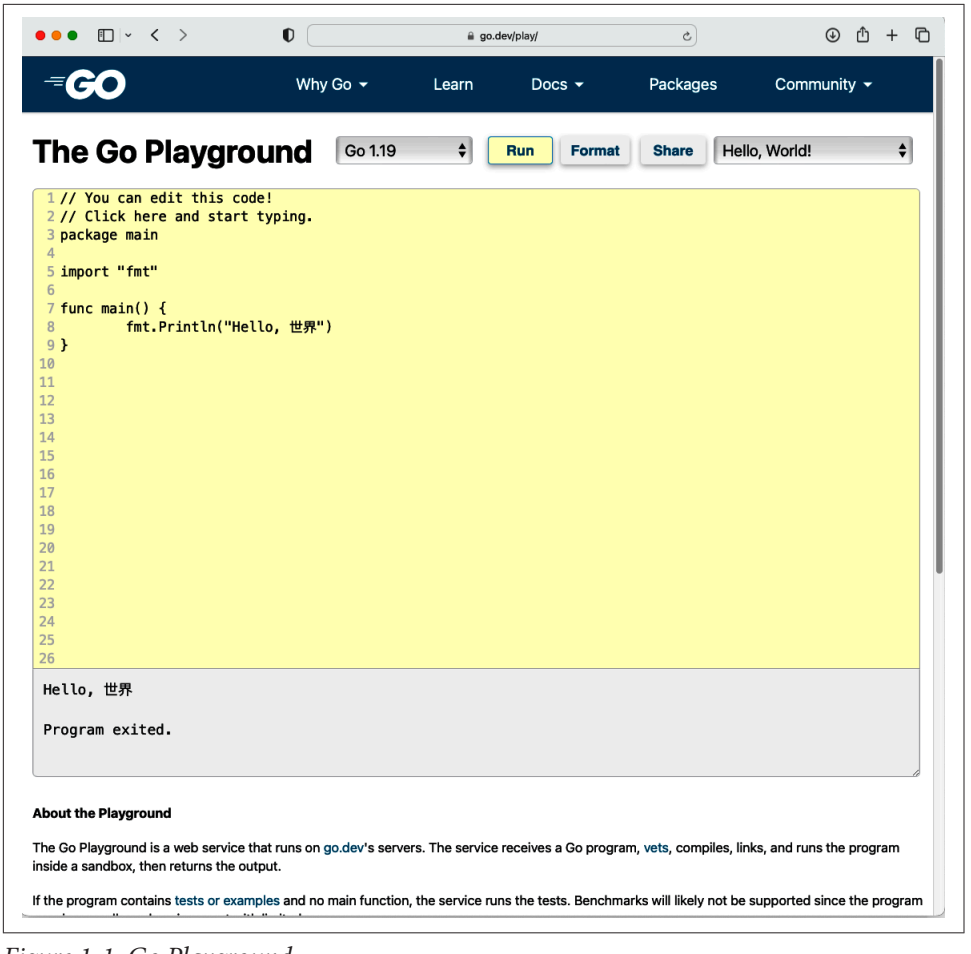

Figure 1-1. Go Playground

1.3 Writing a Hello World Program

Problem

You want to create a simple Hello World program in Go.

Solution

Write the Hello World code in Go, build, and run it.

Discussion

Here's a straightforward Hello World program. Put this in a file named *hello.go*, in a directory named `hello`:

```
package main

import "fmt"

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

The first line defines the package this code runs in. Functions are grouped in packages, and code files in the same packages are all in the same directory. The `main` package is special because it tells Go this should be compiled as an executable program. We also import the `fmt` package, which is part of the standard library.

The `main` package must have a `main` function, which is where the execution of the program starts. In the body of the `main` function, we use the `Println` function in the `fmt` package to print out the string "Hello, World!" The `fmt` package is part of the Go standard library. Go has a pretty good standard library that covers most of what you will typically need. Visit Go's Standard library to find out what the standard library has.

You can run this program immediately by running it from the command line:

```
$ go run hello.go
```

You should see this on your screen:

```
Hello, World!
```

You can also compile it into an executable file by running this command:

```
$ go build
```

This will create an executable file named *hello* (macOS or Linux) or *hello.exe* (Windows) in the same directory. The name *hello* follows the name of the directory it's in. You can change the output name with this command:

```
$ go build -o helloworld
```

This will create the executable file named *helloworld* (macOS or Linux) or *helloworld.exe* (Windows).

1.4 Using an External Package

Problem

You want to import a package from an external library.

Solution

Use the `import` keyword to import an external package.

Discussion

Let's say you want to display the size of a file. You get the exact file size, but it's a relatively large number that is not intuitive to users. You want to easily display the file size without doing much of the mathematics yourself.

You searched the internet and found this interesting third-party, open source package at *https://github.com/dustin/go-humanize*, and it does all that you want. How can you include it and use the functions in the package?

You can do this just like importing a standard library, but instead of using the package name, use the package location. Normally you'd expect the name of the package to be `go-humanize`. However, the package name used in the code itself is `humanize`. This is because the package name, as defined by the author of this package, is `humanize`:

```
package main

import (
        "fmt"

        "github.com/dustin/go-humanize"
)

func main() {
        var number uint64 = 123456789
        fmt.Println("Size of file is", humanize.Bytes(number))
}
```

In many cases, the last directory of the location is the name of the package, but it's not always necessarily so. You can even change the way you call functions in the external package:

```
package main

import (
        "fmt"

        human "github.com/dustin/go-humanize"
)

func main() {
        var number uint64 = 123456789
        fmt.Println("Size of file is", human.Bytes(number))
}
```

Notice that you now use the name `human` when calling the functions. Why does Go allow this? It's because there might be conflicting package names since Go doesn't control how the package is named, nor does it have a centralized repository of packages.

If you try to run this directly, assuming the source code is in a file named *human.go*:

```
$ go run human.go
```

you will see this error message:

```
human.go:6:2: no required module provides package github.com/dustin/go-humanize:
go.mod file not found in the current directory or any parent directory; see
'go help modules'
```

This is because Go doesn't know where to find the third-party package (unlike the standard library packages); you must tell it. To do this, you first need to create a Go module:

```
$ go mod init github.com/sausheong/humanize
```

This creates a Go module with the module path *github.com/sausheong/humanize*, specified in a *go.mod* file. This file provides Go with information on the various third-party packages to include. Then you can get the `go-humanize` package using the `go` tool again:

```
$ go get github.com/dustin/go-humanize
```

This will add the third-party package to the module. To clean up, you can run the following:

```
$ go mod tidy
```

This will clean up the *go.mod* file. You will get back to Go modules in Chapter 2.

1.5 Handling Errors

Problem

You want to take care of errors, which will inevitably occur because things never happen as expected.

Solution

Check if a function returns an error and handle it accordingly.

Discussion

Error handling in Go is important. Go is designed such that you need to check for errors explicitly. Functions that could go wrong will return a built-in type called `error`.

Functions that convert a string to a number (like `ParseFloat` and `ParseInt`) can get into trouble because the string might not be a number, so it will always return an error. For example, in the `strconv` package, functions that convert a number to a string (like `FormatFloat` and `FormatInt`) do not return errors. This is because you are forced to pass in a number, and whatever you pass in will be converted into a string.

Take a look at the following code:

```
func main() {
        str := "123456789"
        num, err := strconv.ParseInt(str, 10, 64)
        if err != nil {
                panic(err)
        }
        fmt.Println("Number is", num)
}
```

The `ParseInt` function takes in a string (and some other parameters) and returns a number `num` and an error `err`. You should inspect the `err` to see if the `ParseInt` function returns anything. If there is an error, you can handle it as you prefer. In this example, you `panic`, which exits the program.

If all goes well, this is what you should see:

```
Number is 123456789
```

If you change `str` to `"abcdefg"`, you will get this:

```
panic: strconv.ParseInt: parsing "abcdefghi": invalid syntax

goroutine 1 [running]:
main.main()
        /Users/sausheong/work/src/github.com/sausheong/gocookbook/ch01_general/
        main.go
        +0xae
exit status 2
```

Of course, you can handle it differently or even ignore it if you want. You'll get in-depth with error handling in Chapter 3.

1.6 Logging Events

Problem

You want to record events that happen during the execution of your code.

Solution

Use the `log` package in the Go standard library to log events.

Discussion

Logging events during code execution gives you a good view of how the code is doing during execution. This is important, especially during long-running programs. Logs help to determine issues with the execution and also the state of the execution. Here is a simple example used earlier:

```
package main

import (
        "fmt"
        "log"
        "strconv"
)

func main() {
        str := "abcdefghi"
        num, err := strconv.ParseInt(str, 10, 64)
        if err != nil {
                log.Fatalln("Cannot parse string:", err)
        }
        fmt.Println("Number is", num)
}
```

When you encounter an error being returned from calling the `strconv.ParseInt` function, you call `log.Fatalln`, which is equivalent to logging the output to the screen and exiting the application. As you can see, logging to the screen also adds the date and time the event occurred:

```
021/11/18 09:19:35 Cannot parse string: strconv.ParseInt: parsing "abcdefghi":
invalid syntax
exit status 1
```

By default, the log goes to standard out, which means it will print to the terminal screen. You can easily convert it to log to a file instead, or even multiple files. More about that in Chapter 4.

1.7 Testing Your Code

Problem

You want to test your code's functionality to ensure it's working the way you want.

Solution

Use Go's built-in testing tool to do functional tests.

Discussion

Go has a useful built-in testing tool that makes testing easier since you don't need to add another third-party library. You'll convert the previous code to a function while leaving your `main` function free:

```
func main() {
}

func conv(str string) (num int64, err error) {
        num, err = strconv.ParseInt(str, 10, 64)
        return
}
```

You'll be doing some testing on this function. To do this, create a file that ends with *_test.go* in the same directory. In this case, create a *conv_test.go* file.

In this file, you can write the various test cases you want. Each test case can correspond to a function that starts with `Test` and takes in a single parameter of type `testing.T`.

You can add as many test cases as you want across all multiple test files, as long as they all end with *_test.go*:

```
package main

import "testing"

func TestConv(t *testing.T) {
        num, err := conv("123456789")
        if err != nil {
                t.Fatal(err)
        }
        if num != 123456789 {
                t.Fatal("Number don't match")
        }
}

func TestFailConv(t *testing.T) {
        _, err := conv("")
```

```
        if err == nil {
                t.Fatal(err)
        }
}
```

Within the test functions, you call the `conv` function that you wanted to test, passing it whatever test data you want. If the function returns an error or the returned value doesn't match what you expect, you call the `Fatal` function, which logs a message and then ends the execution of the test.

Try it: run this from the command line. The flag `-v` is to increase its verbosity so you can see how many test cases are executed and passed:

```
$ go test -v
```

This is what you see:

```
=== RUN   TestConv
--- PASS: TestConv (0.00s)
=== RUN   TestFailConv
--- PASS: TestFailConv (0.00s)
PASS
ok      github.com/sausheong/gocookbook/ch01_general
```

As you can see, all your cases pass. Now make a small change in your `conv` function:

```
func conv(str string) (num int64, err error) {
        num, err = strconv.ParseInt(str, 2, 64)
        return
}
```

Instead of parsing the number as base 10, you use base 2. Rerun the test:

```
=== RUN   TestConv
    general_test.go:8: strconv.ParseInt: parsing "123456789": invalid syntax
--- FAIL: TestConv (0.00s)
=== RUN   TestFailConv
--- PASS: TestFailConv (0.00s)
FAIL
exit status 1
FAIL    github.com/sausheong/gocookbook/ch01_general
```

You see that the `TestConv` test case failed because it no longer returns the expected number. However, the second test case passes because it tests for a failure and it encountered it.

Testing is covered more extensively in Chapter 18.

CHAPTER 2

Module Recipes

2.0 Introduction

Managing dependencies is a crucial part of the software development lifecycle. A package manager is used to automate the downloading, updating, and removing of dependencies. This ultimately ensures that change management is done reliably and that nothing breaks after patches and upgrades.

You can find package managers everywhere. On Linux you have rpm, dpkg, apk, etc., and on macOS, you have Homebrew, MacPorts, Fink, etc. You can even consider the Mac App Store and the Windows Store package managers. Most programming languages also have package managers. For example, Python has pip and conda; Ruby has gems and bundler; PHP has PEAR, Composer, Poetry, and so on. Their features might differ, but at the end of the day, all package managers aim to make developers' lives easier when managing libraries, especially third-party libraries.

Go has an interesting approach to package management. The `go` tool does package management; for example, `go get` will download and install third-party packages. However, before Go 1.11, Go didn't bundle any versioning and dependency management mechanisms. This meant you inadvertently got only the latest version when you got a package.

Sure, there were third-party package and dependency managers like dep and Glide, but they weren't part of Go. That changed in August 2018 when Go 1.11 introduced the concept of modules.

Of course, if your software program is small (you stuff everything into the main package) and if you only use the standard library, versioning is not a big issue. But once your software program becomes large and has many third-party packages, managing it suddenly becomes complex. Versioning becomes critical, and resolving the dependencies can be a nightmare.

Also, third-party packages don't necessarily mean they are managed by someone else; they could also be yours, for example, packages created by your teammates in a larger project.

In Go, a module is a way to group and version packages. It helps developers to manage dependencies. This is critical for most developers who must scale their applications and make them extensible.

The mechanism itself is relatively simple, almost trivial. Go uses a minimal version selection (MVS) algorithm to select the versions. This is defined in a file named *go.mod*. The *go.mod* file specifies the versions of the packages used in the software program—and that's it. Go will not use any version of the package other than the one you specify in *go.mod*. Go modules are also retrofitted into existing tools. This means tools like `go get`, `go build`, `go run`, `go test`, and so on understand modules.

In this chapter, we'll explore how Go modules work and discuss a few important things Go programmers should note.

2.1 Creating a Go Module

Problem

You want to set up your project as a module.

Solution

Run `go mod init` from the command line to create a *go.mod* file.

Discussion

Go modules are simple. Most of the module features in Go are built into existing tools. The only new tool is `go mod`. To set up your project as a module, you run this in your project directory:

```
$ go mod init
```

You will be asked to provide more information if you don't have your `GOPATH` set up or if your project directory is not within the `GOPATH`:

```
go: cannot determine module path for source directory /Users/sausheong/go/src/
github.com/sausheong/gocookbook/ch02_modules (outside GOPATH, module path must
be specified)

Example usage:
        'go mod init example.com/m' to initialize a v0 or v1 module
        'go mod init example.com/m/v2' to initialize a v2 module

Run 'go help mod init' for more information.
```

You can set up your `GOPATH` or follow the suggestions to provide the module path to `go mod init` like this:

```
$ module github.com/sausheong/gocookbook/ch02_modules
```

Once you run `go mod init` successfully, it will create a file named *go.mod* in your project directory. This file defines the module and is central to everything else you do with modules. Your mod file should be something like this:

```
module github.com/sausheong/gocookbook/ch02_modules

go 1.20
```

The first line starts with the `module` directive and describes the module path, which is the location of your project directory. The module path should be a location from which Go tools can download the module, such as the module code's repository location. There is a blank line after that, followed by the minimum version of Go used in this project, specified in the `go` directive.

You might think that the `go` directive will fail compilation if you use a lower version of the language to compile. However, the `go` directive just specifies which language features are available to the code. You will be flagged only if you use an earlier version of Go to compile and your code needs the newer version's features.

2.2 Importing Dependent Packages Into Your Module

Problem

You want to import a third-party package into your module.

Solution

Use the `go get` tool to download the third-party package. The *go.mod* file will be automatically modified to require the new package.

Discussion

One of the cool things about Go modules is how the existing `go` tools have incorporated it seamlessly, and you don't need to change your workflow to accommodate it. Before Go modules, when you wanted to use a third-party package, you needed to use `go get` to download the package before you could add it to your code.

This doesn't change with Go modules. The third-party package you download using `go get` simply gets added to the *go.mod* file. Let's look at an example of a simple web application that uses the `gorilla/mux` third-party package.

Let's say you have run `go mod init` in a project directory, and this *go.mod* file was created:

```
module github.com/sausheong/gocookbook/ch02_modules/hello

go 1.20
```

You also have your program in a *main.go* file:

```
package main

import (
        "log"
        "net/http"
        "text/template"

        "github.com/gorilla/mux"
)

func main() {
        r := mux.NewRouter()
        r.HandleFunc("/{text}", name)
        log.Fatal(http.ListenAndServe(":8080", r))
}

func name(w http.ResponseWriter, r *http.Request) {
        vars := mux.Vars(r)
        t, _ := template.ParseFiles("static/text.html")
        t.Execute(w, vars["text"])
}
```

You will receive an error message if you try running it immediately:

```
$ go run main.go
main.go:8:2: no required module provides package github.com/gorilla/mux; to add
it:
        go get github.com/gorilla/mux
```

This is because you need to download the `gorilla/mux` package before using it:

```
$ go get github.com/gorilla/mux
go: added github.com/gorilla/mux v1.8.0
```

Once you have downloaded the package, the *go.mod* file will be updated. This is the *go.mod* file after getting the `gorilla/mux` package:

```
module github.com/sausheong/gocookbook/ch02_modules/hello

go 1.20

require github.com/gorilla/mux v1.8.0 // indirect
```

Notice the `require` directive requiring the third-party package you just downloaded. The `// indirect` comment means it has been added as an indirect dependency. The `v1.8.0` after the package's module path is the version you use in this project. You didn't specify the version earlier, so Go will take the latest version.

What's the difference between direct and indirect dependencies? Direct dependencies, as the name suggests, are dependencies that are directly imported by the program. The `gorilla/mux` package is directly imported in the source code as in the preceding example, so this *should* be a direct dependency. (The following paragraphs explain why it's not.)

On the other hand, indirect dependencies are dependencies that your third-party packages have. If the package you import doesn't use Go modules and therefore doesn't have a *go.mod* file, these packages will be added to your *go.mod* file by the `require` directive but will be flagged as an indirect dependency.

So why is the `gorilla/mux` package flagged as indirect? When you do a `go get` on any third-party package, it will be added as an indirect dependency first, regardless of whether it's directly imported in any code. After downloading the package, you must run this from the command line to tidy up:

```
$ go mod tidy
```

This will clean things up, and the indirect flag will be removed:

```
module github.com/sausheong/gocookbook/ch02_modules/hello

go 1.20

require github.com/gorilla/mux v1.8.0
```

What if you already downloaded the package? Do you need to download it again every time? The answer is no; when you run `go mod tidy`, Go will figure it out from your source code and add that into your *go.mod* file as a direct dependency:

```
go: finding module for package github.com/gorilla/mux
go: found github.com/gorilla/mux in github.com/gorilla/mux v1.8.0
```

Go will also create another file for you, named *go.sum*. This is what you see when you open it:

```
github.com/gorilla/mux v1.8.0 h1:i40aqfkR1h2SlN9hojwV5ZA91wcXFOvkdNIeFDP5koI=
github.com/gorilla/mux v1.8.0/go.mod h1:DVbg23sWSpFRCP0SfiEN6jmj59UnW/n46BH5rLB7
1So=
```

The *go.sum* file lists the checksum of the dependencies required and the versions. This is a security feature; it's there to make sure that the dependencies are not modified. There are two lines here. The first line contains the checksum of the package's source code, while the second line (the one with `/go.mod`) is the checksum of the package's *go.mod* file. The `h1:` on each line indicates the hash algorithm used (SHA-256).

You should check both the *go.mod* and the *go.sum* files into your source code repository.

Now you can run your program:

```
$ go run main.go
```

Now open up *http://localhost:8080/sausheong* in a browser. You should see the screen shown in Figure 2-1.

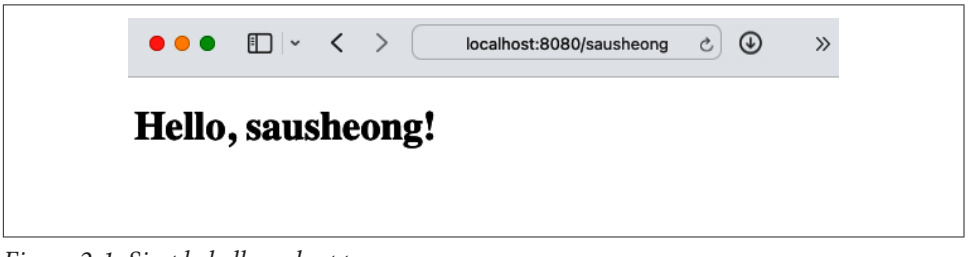

Figure 2-1. Simple hello web app

2.3 Removing Dependent Packages from Your Module

Problem

You want to remove an existing dependent package.

Solution

Remove the dependency on the package in the source code, then run `go mod tidy` to remove the dependency on the package in the *go.mod* and *go.sum* files.

Discussion

Removing a dependent package is simple. You can start from the code by removing the use of the package and the package itself from the list of dependent packages. For example, if you only use the standard library, you can change the code to this:

```
package main

import (
        "log"
        "net/http"
        "text/template"
)

func main() {
        http.HandleFunc("/", name)
        log.Fatal(http.ListenAndServe(":8080", nil))
}

func name(w http.ResponseWriter, r *http.Request) {
        t, _ := template.ParseFiles("static/text.html")
        t.Execute(w, r.URL.EscapedPath()[1:])
}
```

The code no longer uses `gorilla/mux`, but the *go.mod* and *go.sum* files have not been updated. Run `go mod tidy` from the command line, which removes the package from *go.mod* and *go.sum*.

The *go.mod* file will change from this:

```
module github.com/sausheong/gocookbook/ch02_modules/hello

go 1.20

require github.com/gorilla/mux v1.8.0
```

to this:

```
module github.com/sausheong/gocookbook/ch02_modules/hello

go 1.20
```

2.4 Find Available Versions of Third-Party Packages

Problem

You want to know which versions of a third-party package are available.

Solution

For Git repositories, you can use `git ls-remote -t` to list all versions, or `git ls-remote -h` to list all branches in the repository. Alternatively, you can go to the GitHub or GitLab website and view tags or branches.

Discussion

This is not exactly a Go recipe, but it is a useful tip to determine the available package versions.

You need to know the versions and branches available if you want to use releases that are not the latest in the package. To do that, you can use the Git command `ls-remote` with the tags `-t` or `-h` to view the tags and heads, respectively. In Git, tags are often used to mark releases or versions. Heads are the tips of branches, so the heads represent the available branches of the repository.

For example, run this from the command line to list all version releases in `gorilla/mux`:

```
$ git ls-remote -t https://github.com/gorilla/mux.git
```

You will see this output:

```
0eeaf8392f5b04950925b8a69fe70f110fa7cbfc        refs/tags/v1.1
b12896167c61cb7a17ee5f15c2ba0729d78793db        refs/tags/v1.2.0
392c28fe23e1c45ddba891b0320b3b5df220beea        refs/tags/v1.3.0
bcd8bc72b08df0f70df986b97f95590779502d31        refs/tags/v1.4.0
24fca303ac6da784b9e8269f724ddeb0b2eea5e7        refs/tags/v1.5.0
7f08801859139f86dfafd1c296e2cba9a80d292e        refs/tags/v1.6.0
53c1911da2b537f792e7cafcb446b05ffe33b996        refs/tags/v1.6.1
e3702bed27f0d39777b0b37b664b6280e8ef8fbf        refs/tags/v1.6.2
a7962380ca08b5a188038c69871b8d3fbdf31e89        refs/tags/v1.7.0
c5c6c98bc25355028a63748a498942a6398ccd22        refs/tags/v1.7.1
ed099d42384823742bba0bf9a72b53b55c9e2e38        refs/tags/v1.7.2
00bdffe0f3c77e27d2cf6f5c70232a2d3e4d9c15        refs/tags/v1.7.3
75dcda0896e109a2a22c9315bca3bb21b87b2ba5        refs/tags/v1.7.4
98cb6bf42e086f6af920b965c38cacc07402d51b        refs/tags/v1.8.0
```

If you're using GitHub or GitLab or some other SaaS that provides a code repository service, you can also go to the site and view the tags or branches.

2.5 Importing a Specific Version of a Dependent Package Into Your Module

Problem

You want to import a specific version of a dependent package instead of the latest one.

Solution

Use `go get` to download the version you want by adding `@version_number` after the module path.

Discussion

Go modules use the Semantic Versioning (Semver) system for versioning. Semver uses three parts in versioning—major, minor, and patch. For example, in `gorilla/mux` the version number `v1.8.0` means a major version 1, minor version 8, and patch version 0.

To download version `v1.7.4` of `gorilla/mux`, run this from the command line:

```
$ go get github.com/gorilla/mux@v1.7.4
```

You should see a change in version of the `gorilla/mux` package in both the *go.mod* and *go.sum* files.

You can even use specific branches or specific commits:

```
$ go get github.com/gorilla/mux@4248f5cd8717eaea35eded08100714b2b2bac756
```

If you run this from the command line, you will get the following:

```
go: downloading github.com/gorilla/mux v1.7.3-0.20190628153307-4248f5cd8717
go get: downgraded github.com/gorilla/mux v1.7.4 => v1.7.3-0.20190628153307-
4248f5
cd8717
```

Your *go.mod* file will look something like this:

```
module github.com/sausheong/gocookbook/ch02_modules

go 1.20

require github.com/gorilla/mux v1.7.3-0.20190628153307-4248f5cd8717
```

You can go back to the latest version by doing this:

```
$ go get github.com/gorilla/mux@latest
```

You can update the package's latest minor or patch version using the `-u` flag:

```
$ go get -u github.com/gorilla/mux
```

However, for major versions, Go modules use a different path altogether. Starting at v2, the path must end in the major version. For example, since `gorilla/mux` has a major version release, the module path will change to `github.com/gorilla/mux/v2`, and you will use `go get github.com/gorilla/mux/v2` instead.

This convention of using different module paths for major versions is called *semantic import versions*. Semantic import versions let you simultaneously use multiple major package versions for incremental migration when migrating a large codebase.

2.6 Requiring Local Versions of Dependent Packages

Problem

You want to use local versions of the dependent packages.

Solution

Set up Go to use a `vendor` directory by running `go mod vendor`.

Discussion

Local versions are the specific version of the dependent packages that you can use and are a safeguard in case the originals disappear (it happens). Having local versions of the dependent packages can be useful (and not only because you have trust issues).

Run this from the command line to download and keep local versions of the dependent packages:

```
$ go mod vendor
```

This will create a `vendor` subdirectory in your project directory and populate it with the dependencies from your *go.mod* file. It also creates a *vendor/modules.txt* file that contains a listing of the packages you have just vendored and the versions they were copied from.

For example, assuming you have the following *go.mod* file:

```
module github.com/sausheong/gocookbook/ch02_modules/hello

go 1.20

require github.com/gorilla/mux v1.8.0
```

you will find a new `vendor` subdirectory created in your project directory (which is also your module path). If you list the `vendor` subdirectory, you should see something like this:

```
vendor % ls -lR
total 8
drwxr-xr-x  3 sausheong  staff  96 Dec 28 09:38 github.com
-rw-r--r--  1 sausheong  staff  76 Dec 28 09:38 modules.txt

./github.com:
total 0
drwxr-xr-x  3 sausheong  staff  96 Dec 28 09:38 gorilla

./github.com/gorilla:
total 0
drwxr-xr-x  11 sausheong  staff  352 Dec 28 09:38 mux
```

```
./github.com/gorilla/mux:
total 224
-rw-r--r--  1 sausheong  staff    276 Dec 28 09:38 AUTHORS
-rw-r--r--  1 sausheong  staff   1486 Dec 28 09:38 LICENSE
-rw-r--r--  1 sausheong  staff  25363 Dec 28 09:38 README.md
-rw-r--r--  1 sausheong  staff  11227 Dec 28 09:38 doc.go
-rw-r--r--  1 sausheong  staff   2619 Dec 28 09:38 middleware.go
-rw-r--r--  1 sausheong  staff  17677 Dec 28 09:38 mux.go
-rw-r--r--  1 sausheong  staff  10522 Dec 28 09:38 regexp.go
-rw-r--r--  1 sausheong  staff  21706 Dec 28 09:38 route.go
-rw-r--r--  1 sausheong  staff    766 Dec 28 09:38 test_helpers.go
```

If you open *modules.txt*, you can see the vendored packages:

```
# github.com/gorilla/mux v1.8.0
## explicit; go 1.12
github.com/gorilla/mux
```

By default, Go will use the version in the `vendor` directory if your Go version is 1.1.4 and above and your *vendor/modules.txt* file is in sync with your *go.mod* file. If you want to enable vendoring explicitly, you can include the `-mod vendor` flag in the `go` command.

For example, to build the project, you will do this:

```
# go build -mod vendor
```

If you want to disable vendoring explicitly, include the `-mod readonly` or `-mod mod` in the `go` command.

What happens if you add a new dependent package after you have vendored the module? If *go.mod* has changed since *modules.txt* was generated, the `go` command will show an error, and you should update the `vendor` directory again by running `go mod vendor`.

Try it out! Instead of showing text, you want to generate a QR code from the given name using the `go-qrcode` package and display it on the browser.

First, you need to get the package:

```
$ go get github.com/yeqown/go-qrcode
```

This adds a whole bunch of indirect dependent packages to *go.mod*. Remember, you're only using `go-qrcode`; the rest are packages it's dependent on:

```
module github.com/sausheong/gocookbook/ch02_modules/vendoring

go 1.20

require github.com/gorilla/mux v1.8.0

require (
```

```
        github.com/fogleman/gg v1.3.0 // indirect
        github.com/golang/freetype v0.0.0-20170609003504-e2365dfdc4a0 // indirect
        github.com/yeqown/go-qrcode v1.5.10 // indirect
        github.com/yeqown/reedsolomon v1.0.0 // indirect
        golang.org/x/image v0.0.0-20200927104501-e162460cd6b5 // indirect
)
```

You change the code to use `go-qrcode`:

```
package main

import (
        "bytes"
        "encoding/base64"
        "log"
        "net/http"
        "text/template"

        "github.com/gorilla/mux"
        "github.com/yeqown/go-qrcode"
)

func main() {
        r := mux.NewRouter()
        r.HandleFunc("/{text}", name)
        log.Fatal(http.ListenAndServe(":8080", r))
}

func name(w http.ResponseWriter, r *http.Request) {
        vars := mux.Vars(r)
        qrc, _ := qrcode.New(vars["text"])
        var buf bytes.Buffer
        qrc.SaveTo(&buf)
        base64 := base64.StdEncoding.EncodeToString(buf.Bytes())
        t, _ := template.ParseFiles("static/index.html")
        t.Execute(w, base64)
}
```

Once done, use `go mod tidy` to clean up the *go.mod* file, which becomes this, with `qr-code` set up as a direct dependency:

```
module github.com/sausheong/gocookbook/ch02_modules/vendoring

go 1.20

require (
        github.com/gorilla/mux v1.8.0
        github.com/yeqown/go-qrcode v1.5.10
)

require (
        github.com/fogleman/gg v1.3.0 // indirect
        github.com/golang/freetype v0.0.0-20170609003504-e2365dfdc4a0 // indirect
```

```
        github.com/yeqown/reedsolomon v1.0.0 // indirect
        golang.org/x/image v0.0.0-20200927104501-e162460cd6b5 // indirect
)
```

So far, this is the standard modules workflow. However, you will hit this error if you try to build or run in vendored mode:

```
go: inconsistent vendoring in /Users/sausheong/go/src/github.com/sausheong/
    gocookbook/ch02_modules/vendoring:
        github.com/yeqown/go-qrcode@v1.5.10: is explicitly required in go.mod,
        but not marked as explicit in vendor/modules.txt
        github.com/fogleman/gg@v1.3.0: is explicitly required in go.mod, but
        not marked as explicit in vendor/modules.txt
        github.com/golang/freetype@v0.0.0-20170609003504-e2365dfdc4a0: is
        explicitly required in go.mod, but not marked as explicit in vendor/
        modules.txt
        github.com/yeqown/reedsolomon@v1.0.0: is explicitly required in go.mod,
        but not marked as explicit in vendor/modules.txt
        golang.org/x/image@v0.0.0-20200927104501-e162460cd6b5: is explicitly
        required in go.mod, but not marked as explicit in vendor/modules.txt

        To ignore the vendor directory, use -mod=readonly or -mod=mod.
        To sync the vendor directory, run:
                go mod vendor
```

The error message tells you exactly what to do. You need to rerun `go mod vendor`. Once you do that, *modules.txt* is updated. When you run `go build -mod vendor` again, it will work.

If you go to the `vendor` subdirectory, you should see the newly vendored package there. In the interest of space, I won't list the files in the `vendor` subdirectory. The *modules.txt* file should also be updated as well, and you should see all the vendored packages in there:

```
# github.com/fogleman/gg v1.3.0
## explicit
github.com/fogleman/gg
# github.com/golang/freetype v0.0.0-20170609003504-e2365dfdc4a0
## explicit
github.com/golang/freetype/raster
github.com/golang/freetype/truetype
# github.com/gorilla/mux v1.8.0
## explicit; go 1.12
github.com/gorilla/mux
# github.com/yeqown/go-qrcode v1.5.10
## explicit; go 1.17
github.com/yeqown/go-qrcode
github.com/yeqown/go-qrcode/matrix
# github.com/yeqown/reedsolomon v1.0.0
## explicit
github.com/yeqown/reedsolomon
github.com/yeqown/reedsolomon/binary
```

```
# golang.org/x/image v0.0.0-20200927104501-e162460cd6b5
## explicit; go 1.12
golang.org/x/image/draw
golang.org/x/image/font
golang.org/x/image/font/basicfont
golang.org/x/image/math/f64
golang.org/x/image/math/fixed
```

You're not supposed to change the code in the vendored versions of the dependent packages. That could make things pretty messy when managing dependencies and could be a security breach. If you rerun `go mod vendor`, Go will check the integrity of the vendored packages and make sure you're using the version in *go.mod*.

While there are advantages to having vendored dependent packages, it can also be challenging to manage, and you will end up in a bloated repository, with versions appearing everywhere.

2.7 Using Multiple Versions of the Same Dependent Packages

Problem

You want to use multiple versions of the same dependent packages in your code.

Solution

Use the `replace` directive in the *go.mod* file to rename your package.

Discussion

Though it might seem like a very niche requirement, there is sometimes a need to be able to use multiple versions of the same package in your project. In the previous recipe, we talked about one possible use—if you want to migrate a large codebase incrementally but still enable it to work while you're migrating, you might want to use different versions of the same package.

Another possible reason is when you have a dependent package that works only with a specific version of another package. This is more common than you might think because packages developed by different developers move according to their project schedules. You could be in trouble if one of your critical dependent packages still uses an older version of a shared dependent package.

Semantic import versions work well because the major versions require a change in the path. However, this wouldn't work in the case of minor version upgrades. In this case, you can use the `replace` directive to rename a package and use the new name to import it into your code.

First, change the *go.mod* file to rename your packages. Say you're using both v1.8.0 and v1.7.4 for `gorilla/mux`.

You rename these packages accordingly using the `replace` directive:

```
module github.com/sausheong/gocookbook/ch02_modules

go 1.20

replace github.com/gorilla/mux/180 => github.com/gorilla/mux v1.8.0

replace github.com/gorilla/mux/174 => github.com/gorilla/mux v1.7.4
```

Run `go get` to get the package versions using their new names:

```
$ go get github.com/gorilla/mux/174
$ go get github.com/gorilla/mux/180
```

If you open your *go.mod*, you should see the two new packages being *required*:

```
module github.com/sausheong/gocookbook/ch02_modules

go 1.20

replace github.com/gorilla/mux/180 => github.com/gorilla/mux v1.8.0

replace github.com/gorilla/mux/174 => github.com/gorilla/mux v1.7.4

require (
        github.com/gorilla/mux/174 v0.0.0-00010101000000-000000000000 // indirect
        github.com/gorilla/mux/180 v0.0.0-00010101000000-000000000000 // indirect
)
```

Now you can go back to your code and import these two packages:

```
package main

import (
        "log"
        "net/http"
        "text/template"

        mux174 "github.com/gorilla/mux/174"
        mux180 "github.com/gorilla/mux/180"
)

func main() {
        r := mux180.NewRouter()
```

```
        r.HandleFunc("/{text}", name)
        log.Fatal(http.ListenAndServe(":8080", r))
}

func name(w http.ResponseWriter, r *http.Request) {
        vars := mux174.Vars(r)
        t, _ := template.ParseFiles("static/text.html")
        t.Execute(w, vars["text"])
}
```

Note that while this program builds and runs because you are using two separate packages, the `mux174.Vars` will not be able to get the path from the URL.

What happens if you vendor the packages now?

Run `go mod vendor` from the command line. Now open your `vendor` package. You should be able to see two different versions of the package under two different directories as if they are different packages altogether.

CHAPTER 3

Error Handling Recipes

3.0 Introduction

In his *An Essay on Criticism*, Alexander Pope wrote, "to err is human." And since software is written by humans (for now), software errs as well. Just like human errors, it's about how gracefully we can recover from them. That's what error handling is all about—how we recover when our program gets into a situation we did not expect or cater to in its normal flow.

Programmers often treat error handling as tedious work and an afterthought. That's generally an error in itself. Just as testing and error handling should be top of mind, recovering from the error should be part of good software design. In Go, error handling is treated pretty seriously, though unconventionally. Go has the `errors` package in the standard library that provides many functions to manipulate errors, but most error handling in Go is built into the language or is part of the idiomatic way of programming in Go. This chapter covers some basic ideas in error handling in Go.

Errors Are Not Exceptions

In programming languages like Python and Java, error handling is done through exceptions. An exception is an object that represents an error, and whenever something goes wrong, you can throw an exception. The calling function usually has a `try` and `catch` (or `try` and `except` in Python) that handles anything that goes wrong.

Go does this slightly differently (or entirely differently, depending on how you look at it). Go doesn't have exception handling. Instead of exceptions, it has errors. An `error` is a built-in type that represents an unexpected condition. Instead of throwing an exception, you will create an error and return it to the calling function. Isn't this like saying the *Odyssey* wasn't written by Homer but by another Greek named Homer, who also lived 2,800 years ago?

Well, not exactly. Functions do not return exceptions at all; you don't know if any exceptions will be returned or what kind of exceptions they will be (well, sometimes you do, but that's a different story). You have to round it up using `try` and `catch`. Exceptions are thrown only when there are problems (that's why they are called exceptions), and you can wrap around potentially exception-causing statements with the same `try` and `catch`. On the other hand, errors are deliberately returned by the function to be inspected by the calling function and dealt with individually.

As a result, programmers familiar with exceptions find error handling in Go particularly tedious. Instead of using a wide net to catch exceptions in a series of statements, you are expected to inspect the returning errors each time and deal with these errors individually. Of course, you can also ignore the errors altogether, though idiomatically you are expected to take errors seriously and deal with them each time you get one. The only exception to this is if you don't care about the returning results at all.

3.1 Handling Errors

Problem

You want to handle an unexpected condition in your code.

Solution

If you're writing a function, return an `error` along with the return value (if any). If you're calling a function, inspect the error returned; if it is not nil, handle it accordingly.

Discussion

There are two ways you need to deal with errors: when writing a function and when calling a function.

Writing a function

Go represents errors with the `error` built-in error type, which is an interface. The rule of thumb when writing functions is that if there is any way the function will fail, you need to return an error, along with whatever return value your function returns. This is possible because Go allows multiple return values. By convention, `error` is the last return value; for example, this function allows you to guess a number:

```
func guess(number uint) (answer bool, err error) {
    if number > 99 {
        err = errors.New("Number is larger than 100")
    }
    // check if guess is correct
    return answer, err
}
```

The input to the function should be less than 100, and the function should return a true or false, indicating whether or not the guess is correct. If the input number is larger than 100, you want to flag an error, which is what you do here by creating a new error using the `errors.New` function:

```
err = errors.New("Number is larger than 100")
```

There are other ways to create a new error, though. Another common way of creating a new error is in the common `fmt` package using the `Errorf` function:

```
err = fmt.Errorf("Number is larger than 100")
```

The big difference between the two in this example is trivial—`fmt.Errorf` allows you to format the string, like the `Printf`, `Sprintf` `Sscanf`, and similar functions in the `fmt` package, while `errors.New` just creates an error with a string. There's a bit more to this in `fmt.Errorf`, though, because it can allow you to wrap an error around another error (see Recipe 3.4).

Calling a function

Another Go rule of thumb is "don't ignore errors." The standard way to handle errors in Go is relatively straightforward—you deal with it just like any other return value. The following example is taken from Recipe 1.5:

```
str := "123456789"
num, err := strconv.ParseInt(str, 10, 64)
if err != nil {
   // handle the error
}
```

The function `strconv.ParseInt` returns two values—the first is the converted integer, and the second is the error. You should inspect the error, and if the error is nil, all is well, and you can continue with the program flow. If it's not nil, then you should handle it. In the example in Recipe 1.5, you called `panic` with the error, discussed later in the chapter, but you can handle it whichever way you want, including ignoring it. You can, of course, also deliberately choose to ignore errors like this:

```
num, _ := strconv.ParseInt(str, 10, 64)
```

Here you're assigning the returned error to the underscore (_), which means you're ignoring it. In either case, it becomes clear that you are deliberately ignoring returned errors. Go cannot stop you if you're not handling the error. However, a linter, your IDE, or code review (if you're on a team) will quickly bring this error to the surface, along with your laziness in handling it.

Why is error handling done this way in Go?

You might wonder why Go does it this way instead of using exceptions like many other languages. Exceptions seem easier to handle because you can group statements and handle them together. Go forces you to handle errors with each function call, which can be tedious.

However, exceptions can also be easily missed unless you have a `try` and `catch`. In addition, if you're wrapping a bunch of statements with `try` and `catch` it's easy to miss handling specific errors, and it can also be confusing if you bunch too many statements together.

The other benefit of using errors instead of exceptions is that the returned error is a value you can use just like any other value in your normal flow. While you can also process exceptions, they are constructs in the `try` and `catch` loop, which is not in your normal flow. It seems like a trivial point, but it's an important one because doing so makes handling errors an integral part of writing your code instead of making it optional.

3.2 Simplifying Repetitive Error Handling

Problem

You want to reduce the number of lines of repetitive error-handling code.

Solution

Use helper functions to reduce the number of lines of repetitive error-handling code.

Discussion

One of the most frequent complaints about Go's error handling, especially from newcomers, is that it's tedious to do repetitive checks. Let's take, for example, this piece of code that opens a JSON file to read and unmarshal to a struct:

```
func unmarshal() (person Person) {
        r, err := http.Get("https://swapi.dev/api/people/1")
        if err != nil {
                // handle error
        }
```

```
        defer r.Body.Close()

        data, err := io.ReadAll(r.Body)
        if err != nil {
                // handle error
        }

        err = json.Unmarshal(data, &person)
        if err != nil {
                // handle error
        }
        return person
}
```

You can see three sets of error handling: one when you call `http.Get` to get the API response into an `http.Response`, then when you call `io.ReadAll` to get the JSON text from the `http.Response`, and finally to unmarshal the JSON text into the `Person` struct. Each of these calls is a potential point of failure, so you need to handle errors that result from those failures.

However, these error-handling routines are similar to each other and, in fact, repetitive. How can you resolve this? There are several ways, but the most straightforward is using helper functions:

```
func helperUnmarshal() (person Person) {
        r, err := http.Get("https://swapi.dev/api/people/1")
        check(err, "Calling SW people API")
        defer r.Body.Close()

        data, err := io.ReadAll(r.Body)
        check(err, "Read JSON from response")

        err = json.Unmarshal(data, &person)
        check(err, "Unmarshalling")
        return person
}

func check(err error, msg string) {
        if err != nil {
                log.Println("Error encountered:", msg)
        // do common error-handling stuff
        }
}
```

The helper function here is the `check` function that takes in an error and a string. Besides logging the string, you can also put all the common error-handling stuff that you want to do into the function. Instead of a string, you can also take in a function as a parameter and execute the function if an error is encountered.

Of course, this is only one possible type of helper function; here's another one. You will use a pattern found in another package in the standard library this time. In the

text/template package, you can find a helper function called template.Must that wraps around functions that return (*Template, error). If a function returns a non-nil error, then Must panics. You can similarly create something like this to wrap around other function calls:

```
func must(param any, err error) any {
        if err != nil {
                // handle errors
        }
        return param
}
```

Because it takes any single parameter (using any) and returns a single value (also using any), you can use this for any function that returns a single value along with an error. For example, you can convert your earlier unmarshal function to something like this:

```
func mustUnmarshal() (person Person) {
        r := must(http.Get("https://swapi.dev/api/people/1")).(*http.Response)
        defer r.Body.Close()
        data := must(io.ReadAll(r.Body)).([]byte)
        must(nil, json.Unmarshal(data, &person))
        return person
}
```

Note that in the first line of the mustUnmarshal function, the function call http.Get returns two values, which are used as parameters in the must function. In the second use of the must function, the json.Unmarshal function returns an error only.

The code is more concise, but at the same time, it also makes the code more unreadable, so this kind of helper function should be used sparingly.

3.3 Creating Customized Errors

Problem

You want to create custom errors to communicate more information about the error encountered.

Solution

Create a new string-based error or implement the error interface by creating a struct with an Error method that returns a string.

Discussion

There are different ways of creating errors (in a good way).

Using a string-based error

The simplest way to implement a customized error is to create a new string-based error. You can use either the `errors.New` function, which creates an error with a simple string, or the `fmt.Errorf` function, which allows you to include formatting for the error string.

The `errors.New` function is very straightforward:

```
err := errors.New("Syntax error in the code")
```

The `fmt.Errorf` function, like many of the functions in the `fmt` function, allows for formatting within the string:

```
err := fmt.Errorf("Syntax error in the code at line %d", line)
```

Implementing the error interface

The `builtin` package contains all the definitions of the built-in types, interfaces, and functions. One of the interfaces is the `error` interface:

```
type error interface {
    Error() string
}
```

As you can see, any struct with a method named `Error` that returns a string is an error. So if you want to define a custom error to return a custom error message, implement a struct and add an `Error` method. For example, let's say you are writing a program used for communications and want to create a custom error to represent an error during communications:

```
type CommsError struct{}

func (m CommsError) Error() string {
        return "An error happened during data transfer."
}
```

You want to provide information about where the error came from. To do this, you can create a custom error to provide the information. Of course, you usually wouldn't override `Error`; you can add fields and other methods to your custom error to carry more information:

```
type SyntaxError struct {
        Line int
        Col  int
}

func (err *SyntaxError) Error() string {
        return fmt.Sprintf("Error at line %d, column %d", err.Line, err.Col)
}
```

When you get such an error, you can typecast it using the "comma, ok" idiom (because if you typecast it and it's not that type, it will panic), and extract the additional data for your processing:

```
if err != nil {
        err, ok := err.(*SyntaxError)
        if ok {
                // do something with the error
        } else {
                // do something else
        }
}
```

3.4 Wrapping an Error with Other Errors

Problem

You want to provide additional information and context to an error you receive before returning it as another error.

Solution

Wrap the error you receive with another error you create before returning it.

Discussion

Sometimes you will get an error, but instead of just returning that, you want to provide additional context before returning the error. For example, if you get a network connection error, you might want to know where in the code that happened and what you were doing when it happened.

Of course, you can simply extract the information, create a new customized error with the additional information, and return that. Alternatively, you can also wrap the error with another error and return it, passing it up the call stack while adding additional information and context.

There are a couple of ways to wrap errors. The easiest is to use `fmt.Errorf` again and provide an error as part of the parameter:

```
err1 := errors.New("Oops something happened.")
err2 := fmt.Errorf("An error was encountered - %w", err1)
```

The `%w` verb allows you to place an error within the format string. In the example, `err2` wraps `err1`. But how do you extract `err1` out of `err2`?

The `errors` package has an `Unwrap` function that does precisely this:

```
err := errors.Unwrap(err2)
```

This will give you back `err1`.

Another way of wrapping an error with an error is to create a customized error struct like this:

```
type ConnectionError struct {
        Host string
        Port int
        Err  error
}

func (err *ConnectionError) Error() string {
        return fmt.Sprintf("Error connecting to %s at port %d", err.Host,
        err.Port)
}
```

Remember, to make it an error, the struct should have an `Error` method. To allow the struct to be unwrapped, you need to implement an `Unwrap` function:

```
func (err *ConnectionError) Unwrap() error {
        return err.Err
}
```

3.5 Inspecting Errors

Problem

You want to check for specific errors or specific types of errors.

Solution

Use the `errors.Is` and `errors.As` functions. The `errors.Is` function compares an error to a value and the `errors.As` function checks if an error is of a specific type.

Discussion

The `errors.Is` and `errors.As` functions are operators that work on errors. They help us to inspect errors and figure out what the errors are.

Using errors.Is

The `errors.Is` function is essentially an equality check. Let's say you define a set of customized errors in your codebase—for example, `ApiErr`—which happens when a connection to an API encounters an error:

```
var ApiErr error = errors.New("Error trying to get data from API")
```

Elsewhere in your code, you have a function that returns this error:

```
func connectAPI() error {
        // some other stuff happening here
        return ApiErr
}
```

You can use `errors.Is` to check if the error returned is `ApiErr`:

```
err := connectAPI()
if err != nil {
        if errors.Is(err, ApiErr) {
                // handle the API error
        }
}
```

You can also verify if `ApiErr` is somewhere along the chain of wrapped errors. Take the example of a `connect` function that returns a `ConnectionError` that wraps around `ApiErr`:

```
func connect() error {
        return &ConnectionError{
                Host: "localhost",
                Port: 8080,
                Err:  ApiErr,
        }
}
```

This code still works because `ConnectionError` wraps around `ApiErr`:

```
err := connect()
if err != nil {
        if errors.Is(err, ApiErr) {
                // handle the API error
        }
}
```

Using errors.As

The `errors.As` function allows you to check for a specific type of error. Continue with the same example, but this time around, you want to check if the error is of the type `ConnectionError`:

```
err := connect()
if err != nil {
        var connErr *ConnectionError
        if errors.As(err, &connErr) {
                log.Errorf("Cannot connect to host %s at port %d", connErr.Host,
                connErr.Port)
        }
}
```

You can use `errors.As` to check if the returned error is a `ConnectionError` by passing it the returned error and a variable of type `*ConnectionError`, named `connErr`. If it turns out to be so, `errors.As` will then assign the returned error into `connErr`, and you can process the error simultaneously.

3.6 Handling Errors with Panic

Problem

You want to report an error that causes your program to halt.

Solution

Use the built-in `panic` function to stop the program.

Discussion

Sometimes your program will encounter an error that makes it unable to continue. In this case, you want to stop the program. Go provides a built-in function called `panic` that takes in a single parameter of any type and stops the normal execution of the current goroutine.

When a function calls `panic`, the normal execution of the function stops immediately, and any deferred actions (anything that you call that starts with `defer`) are executed before the function returns to its caller.

The calling function will also panic and stop normal execution and execute deferred actions as well. This bubbles up until the entire program exits with a nonzero exit code.

Let's take a closer look at this. You create a normal flow of functions where `main` calls `A`, `A` calls `B`, and `B` calls `C`:

```
package main

import "fmt"

func A() {
        defer fmt.Println("defer on A")
        fmt.Println("A")
        B()
        fmt.Println("end of A")
}

func B() {
        defer fmt.Println("defer on B")
        fmt.Println("B")
        C()
```

```
        fmt.Println("end of B")
}

func C() {
        defer fmt.Println("defer on C")
        fmt.Println("C")
        fmt.Println("end of C")
}

func main() {
        defer fmt.Println("defer on main")
        fmt.Println("main")
        A()
        fmt.Println("end of main")
}
```

The execution results are as follows, which is the expected flow:

```
% go run main.go
main
A
B
C
end of C
defer on C
end of B
defer on B
end of A
defer on A
end of main
defer on main
```

What happens if you call `panic` in `C` between the two `fmt.Println` statements like this?

```
func C() {
        defer fmt.Println("defer on C")
        fmt.Println("C")
        panic("panic called in C")
        fmt.Println("end of C")
}
```

When `C` calls `panic` in the middle of the execution, `C` stops immediately and executes the deferred code within its scope. After that, it bubbles up to the caller `B`, which also stops immediately, executes the deferred code within its scope, and returns to its calling function, `A`. The same happens to `A`, which bubbles up the `main` function, which executes the deferred code within its scope. Since that's the end of the chain, it will print out the `panic` parameter.

This is what you should see if you run this code on the terminal:

```
% go run main.go
main
A
B
C
defer on C
defer on B
defer on A
defer on main
panic: panic called in C
```

As you can see from the results, the rest of the code in all the functions never gets executed, but all the deferred code gets executed before `panic` exits with a parameter.

3.7 Recovering from Panic

Problem

One of your goroutines has an error and cannot continue, and a `panic` is called, but you don't want to stop the rest of the program.

Solution

Use the built-in `recover` function to stop the panic. This works only in deferred functions.

Discussion

In Recipe 3.6, you saw how `panic` stops the normal execution of code, runs the deferred code, and bubbles up until the program terminates. Calling `panic` doesn't always mean the program terminates. Sometimes you don't want `panic` to end the program. In this case, you can use the built-in `recover` function to stop `panic` and continue the execution of the program.

But why would you want that? There could be a few reasons. You could be using a package that panics whenever it encounters something it cannot recover from. However, that doesn't mean you want your program to terminate (maybe it's OK for you even if that part of the code cannot continue). Or you simply want to stop the execution of a goroutine without killing off the main program; for example, if your web application is running, you don't want a panicked handler function to shut down the whole server.

Whichever case it is, `recover` can work only if you use it within a `defer`. This is because when a function calls `panic`, everything else stops working except for deferred code.

Here is the example from Recipe 3.6:

```
package main

import "fmt"

func A() {
        defer fmt.Println("defer on A")
        fmt.Println("A")
        B()
        fmt.Println("end of A")
}

func B() {
        defer dontPanic()
        fmt.Println("B")
        C()
        fmt.Println("end of B")
}

func C() {
        defer fmt.Println("defer on C")
        fmt.Println("C")
        fmt.Println("end of C")
}

func main() {
        defer fmt.Println("defer on main")
        fmt.Println("main")
        A()
        fmt.Println("end of main")
}

func dontPanic() {
        err := recover()
        if err != nil {
                fmt.Println("panic called but everything's ok now:", err)
        } else {
                fmt.Println("defer on B")
        }
}
```

You added a new function named `dontPanic` that is called during a `defer` call in `B`. In `dontPanic`, you call the built-in function `recover`. Under normal circumstances, `recover` returns nil, and the usual deferred code is run, printing out "defer on B":

```
% go run main.go
main
A
B
C
end of C
```

```
defer on C
end of B
defer on B
end of A
defer on A
end of main
defer on main
```

If a panic happens, `recover` will return the parameter passed to `panic` and stop `panic` from continuing. To see how this works, add a `panic` into `C`:

```
func C() {
        defer fmt.Println("defer on C")
        fmt.Println("C")
        panic("panic called in C")
        fmt.Println("end of C")
}
```

Run the program again and see what happens:

```
% go run main.go
main
A
B
C
defer on C
panic called but everything's ok now: panic called in C
end of A
defer on A
end of main
defer on main
```

When `panic` is called in `C`, the deferred code in `C` kicks in without running the rest of the code in `C`, and bubbles up to `B`. `B` stops running the rest of the code and starts running the deferred code, which calls `dontPanic`. `dontPanic` calls `recover`, which returns the parameter passed to the `panic` called in `C`, and the recovery code is run. This prints out "panic called, but everything's ok now:" and the parameter passed to `panic`, which is "panic called in C". The `panic` stops at this point. Normal execution of `B` doesn't happen, but when `B` returns to `A`, all is well, and the normal execution flow of the code continues.

3.8 Handling Interrupts

Problem

Your program receives an interrupt signal from the operating system (for example, if the user presses Ctrl-C), and you want to clean up and exit gracefully.

Solution

Use a goroutine to monitor the interrupt using the `os/signal` package. Place your clean-up code in the goroutine.

Discussion

Signals are messages sent to running programs to trigger certain behaviors within the program. Signals are asynchronous and can be sent by the operating system or other running programs. When a signal is sent, the operating system interrupts the running program to deliver the signal. If the process has a signal handler for the signal, that handler will be executed. Otherwise, a default handler will be executed.

On the command line, certain key combinations like Ctrl-C trigger a signal (in this case, Ctrl-C sends the `SIGINT` signal) to the program running in the foreground. The `SIGINT` signal or *signal interrupt* is a signal that interrupts the running program and causes it to terminate.

You can capture such signals in Go using the `os/signal` package.

Here's how you can do this:

```
ch := make(chan os.Signal)
signal.Notify(ch, os.Interrupt)

go func() {
        <-ch
        // clean up before graceful exit
        os.Exit(0)
}()
```

First, you create a channel `ch` to send the signals. Then you use the `signal.Notify` function to relay incoming signals to `ch`. The first parameter of `signal.Notify` is the channel, and the second parameter is variadic, which means you can pass in none or more parameters. In this case, you pass in the various signals you want to capture. In the preceding example code, you want to relay `os.Interrupt`, which is a `syscall.SIGINT` or Ctrl-C. If no parameters are passed in, all signals will be relayed to the channel.

After you have set things up, you spin up a goroutine where you wait for the signal to come in by receiving from `ch`. This causes the goroutine to block until a signal is sent on it. Once a signal comes in, you continue the goroutine, executing whichever clean-up routine you want before gracefully exiting the program.

CHAPTER 4

Logging Recipes

4.0 Introduction

Logging is the act of recording events that occur during the running of a program. It is often an undervalued activity in programming because it is additional work that has little immediate payback for the programmer.

During the normal operations of a program, logging is an overhead, taking up processing cycles to write to a file, database, or even to the screen. In addition, unmanaged logs can cause problems. The classic case of logfiles getting so big that they take up all the available disk space and crash the server is too real and happens too often.

However, when something happens, and you want to find out the sequence of events that led to it, logs become an invaluable diagnostic resource. Logs can also be monitored in real time, and alerts can be sent out when needed.

4.1 Writing to Logs

Problem

You want to log events that happen during the execution of your code.

Solution

Use the `log` package in the standard library to log events.

Discussion

Go provides a `log` package in the standard library that you can use to log events while the program is running. It has a default implementation that writes to standard error and adds a timestamp. This means you can use it out of the box for logging without configuration or setup if you're looking to log to standard error.

The `log` package provides several functions that allow you to write logs. In particular, there are three sets of functions:

Print
: Prints the logs to the logger

Fatal
: Prints to the logger and calls `os.Exit` with an exit code of 1

Panic
: Prints to the logger and calls `panic`

Each set comes in a triplet of functions; for example, `Print` has `Print` and `Printf`, which allow formatting, and `Println` adds a newline after printing.

Here is an example with `Print`:

```
func main() {
        str := "abcdefghi"
        num, err := strconv.ParseInt(str, 10, 64)
        if err != nil {
                log.Println("Cannot parse string:", err)
        }
        fmt.Println("Number is", num)
}
```

In this example, when you encounter an error, you call the `Println` function that prints to the standard logger. You will see this on the command line when you run the program:

```
% go run main.go
2022/01/23 18:39:06 Cannot parse string: strconv.ParseInt: parsing "abcdefghi":
 invalid syntax
Number is 0
```

You will also see that the program doesn't stop and continues to the final statement of the program. In other words, `Println` simply logs and continues. How is it different from `fmt.Println`? It's not, really—the only thing it adds to the line is the date.

Next, take a look at `Fatal`. You change just one line of the code to use `Fatalln`:

```
func main() {
        str := "abcdefghi"
        num, err := strconv.ParseInt(str, 10, 64)
```

```
        if err != nil {
                log.Fatalln("Cannot parse string", err)
        }
        fmt.Println("Number is", num)
}
```

When you run it, you should see this:

```
% go run main.go
2022/01/23 18:42:10 Cannot parse string strconv.ParseInt: parsing "abcdefghi":
invalid syntax
exit status 1
```

Notice that the final statement isn't executed, and the program ends with exit code 1. Exit code 1 is a catch-all for general errors, meaning something went wrong with the program, and that's why it has to exit.

Finally, you can use `Panic`. When you call the `panic` built-in function, it will halt the current goroutine, run the deferred code, and return to the calling function, triggering another `panic`, which bubbles up eventually to `main` and finally exits. Refer to Recipe 3.7 for more on the built-in `panic` function:

```
func main() {
        str := "abcdefghi"
        num := conv(str)
        fmt.Println("Number is", num)
}

func conv(str string) (num int64) {
        defer fmt.Println("deferred code in function conv")
        num, err := strconv.ParseInt(str, 10, 64)
        if err != nil {
                log.Panicln("Cannot parse string", err)
        }
        fmt.Println("end of function conv")
        return
}
```

The `Panicln` function prints to the standard logger and panics. When you run the code, this is what you will see:

```
% go run main.go
2022/01/23 18:48:20 Cannot parse string strconv.ParseInt: parsing "abcdefghi":
invalid syntax
deferred code in function conv
panic: Cannot parse string strconv.ParseInt: parsing "abcdefghi": invalid syntax
...
exit status 2
```

The deferred code in the `conv` function runs, but the final statement in the program doesn't. Interestingly you see exit code 2, which is technically inaccurate because

traditionally, exit code 2 means something like "incorrect arguments." As of Go version 1.17.6 this minor bug is still in the backlog, waiting to be fixed.

4.2 Change What Is Being Logged by the Standard Logger

Problem

You want to change what the standard logger logs.

Solution

Use the `SetFlags` function to set flags and add fields to each log line.

Discussion

The default behavior of the standard logger adds the date and time fields to each line of the log. For example, with this line of code:

```
log.Println("Some event happened")
```

you will see this on the screen:

```
2022/01/24 10:46:44 Some event happened
```

The `log` package allows you to add information along with the default date and time fields. You can add these fields using the `SetFlag` function. The fields that are provided include:

Date
: The date in local time zone

Time
: The time in local time zone

Microseconds
: The microsecond resolution of the time field

UTC
: Use UTC time zone instead of local time zone if date or time fields are set

Long file
: The full filename and line number

Short file
: The filename and the line number

Message prefix position
: Move the prefix (from `SetPrefix`) from the beginning of the line to before the start of the message

Here are some examples. You start by setting only the date field in the log:

```
log.SetFlags(log.Ldate)
log.Println("Some event happened")
```

This produces:

```
2022/01/24 Some event happened
```

If you want to add the time with microsecond details, you do this:

```
log.SetFlags(log.Ldate | log.Lmicroseconds)
log.Println("Some event happened")
```

Using the `or` operator on the flags, you set up the various fields to use with the log. Here's the result from before:

```
2022/01/24 20:43:54.595365 Some event happened
```

The file fields are interesting because you can use them to tell you where the problems lie in the code through the logs:

```
log.SetFlags(log.Ldate | log.Lshortfile)
log.Println("Some event happened")
```

It gives you additional information about the filename and the line where the problem occurred:

```
2022/01/24 20:51:02 logging.go:20: Some event happened
```

4.3 Logging to File

Problem

You want to log events to a logfile instead of standard error.

Solution

Use the `SetOutput` function to set the log to write to a file.

Discussion

So far, you've learned about writing the logs to standard error, mainly on the screen if you run it in the command line. What if you want to write it to a logfile, which is common in most cases?

The answer is pretty simple. You use `SetOutput` to redirect the output to a file.

First, look at the logfile. You want to open a new file for create or append, and it's for write only:

```
file, err := os.OpenFile("app.log", os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
if err != nil {
    log.Fatal(err)
}
defer file.Close()
```

You call `SetOutput` with the file as the parameter, then continue to log:

```
log.SetOutput(file)
log.Println("Some event happened")
```

The output will be written to the *app.log* file instead of the screen.

As you probably realize from the code, setting up the logs to be written to file should be done once. What if you want to write to the screen and the file simultaneously? You could reset the log output each time (don't do this) or maybe create two loggers, one for standard error and another for a file, then call the loggers in separate statements.

Or you can use Go's `MultiWriter` function in the `io` package, which creates a writer that duplicates its writes to all the provided writers:

```
file, err := os.OpenFile("app.log", os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
if err != nil {
    log.Fatal(err)
}
defer file.Close()
writer := io.MultiWriter(os.Stderr, file)
log.SetOutput(writer)
log.Println("Some event happened")
log.Println("Another event happened")
```

Doing this will write to both the screen and the file simultaneously. In fact, you can write to more than two writers!

4.4 Using Log Levels

Problem

You want to log events according to log levels.

Solution

Use the `New` function to create a logger, one for each log level, and then use those loggers accordingly.

Discussion

Log data is usually pretty large. You can use log levels to make it more manageable and determine the priority of the events. Log levels indicate the event's severity, indicating the event's importance. It's a simple mechanism—you look at the higher-severity log levels first, filtering off lower-level logs and reducing alert fatigue. Examples of log levels from high to low are:

- Fatal
- Error
- Warn
- Info
- Debug

To set up log levels for your logs, you can add the level to each line of the log. The most straightforward way of doing this is to use the `SetPrefix` function:

```
log.SetPrefix("INFO ")
log.Println("Some event happened")
```

If you call the `SetPrefix` function with the log level as the prefix, you will set the log level at the beginning of the line:

```
INFO 2022/01/26 00:48:15 Some event happened
```

Of course, the problem is that each time you want to add a log line with a different log level from the previous line, you need to call `SetPrefix` again. That is not a feasible solution.

Another method is to create new loggers upfront, with each logger representing a single log level:

```
var (
        info  *log.Logger
        debug *log.Logger
)

func init() {
        info = log.New(os.Stderr, "INFO\t", log.LstdFlags)
        debug = log.New(os.Stderr, "DEBUG\t", log.LstdFlags)
}
```

To do this, you use the `New` function, which returns a logger, but you can also set the prefix and the fields to add to the log.

All you need to do to log events with different log levels is to use the appropriate loggers:

```
info.Println("Some informational event happened")
debug.Println("Some debugging event happened")
```

This is what will appear on the screen:

```
INFO    2022/01/26 00:53:03 Some informational event happened
DEBUG   2022/01/26 00:53:03 Some debugging event happened
```

You can also turn off logging for specific log levels to reduce the logfile size. For example, when you're developing the program you can log debug events, but once you run in production, you no longer want to do that. A common practice is to use environment variables to indicate if you are running in development or production.

Environment variables are named values that are part of the environment in which programs run. Environment variables are set at the command line, and programs can access these variables during runtime.

For example, for Unix-based systems like Linux and macOS, you can set an environment variable `ENV` like this:

```
$ export ENV=development
```

To get the environment variable at the command line, you can use `echo` and add a `$` in front of the variable name:

```
$ echo $ENV
```

Similarly, in Windows systems, you can set the environment variable this way:

```
$ set ENV=development
```

To get the environment variable in Windows, you can use `echo` and add a `%` before and after the variable name:

```
$ echo %ENV%
```

Retrieving the environment variable in Go is simple. You can use the `os.Getenv` function, passing it the environment variable name, and you will get the environment variable value. Using the same previous example, let's set the `ENV` environment variable to `production`:

```
$ export ENV=production
```

If you run this code, you will see that the debug event is not printed:

```
info.Println("Some informational event happened")
if os.Getenv("ENV") != "production" {
        debug.Println("Some debugging event happened")
}
```

If you switch to the development environment, the debug event is printed again.

At the start of this recipe, you learned that one of the reasons to use log levels is to prioritize and filter off certain log levels. You can do this easily with a Unix-based system using the `grep` command.

Say you have a file named *logfile.log* with the following entries:

```
INFO    2023/01/06 00:21:32 Some informational event happened
DEBUG   2023/01/06 00:21:32 Some debugging event happened
INFO    2023/01/06 00:21:35 Another informational event happened
WARN    2023/01/06 00:23:35 A warning event happened
WARN    2023/01/06 00:33:11 Another warning event happened
ERROR   2023/01/06 00:33:11 An error event happened
```

You want to look at all error events first so you can use `grep` to filter out only error events:

```
$ grep "^ERROR" logfile.log
```

You will see only the error event. The `^` in front of `grep` means you just want to see the lines that start with `ERROR`:

```
ERROR   2023/01/06 00:33:11 An error event happened
```

What if you want to see all log events except for debug events? You can just exclude the debug events using the `v` flag in `grep`:

```
$ grep -v "^DEBUG" logfile.log
```

This will result in all events being shown except for debug events:

```
INFO    2023/01/06 00:21:32 Some informational event happened
INFO    2023/01/06 00:21:35 Another informational event happened
WARN    2023/01/06 00:23:35 A warning event happened
WARN    2023/01/06 00:33:11 Another warning event happened
ERROR   2023/01/06 00:33:11 An error event happened
```

Using `grep` is only the beginning. `grep` is a powerful tool, but there are many other log analysis tools you can use.

4.5 Logging to the System Log Service

Problem

You want to log in to the system log instead of your logfiles.

Solution

Use the `log/syslog` package to write to syslog.

Discussion

Syslog is a standard network-based logging protocol. It has long been the de facto standard for logging system events and was created by Eric Allman in the 1980s as part of the Sendmail project. The protocol was documented in RFC 3164 by the Internet Engineering Task Force (IETF). Subsequently, IETF standardized it in RFC 5424.

A syslog message (as in RFC 3164) consists of three parts:

Priority
: Includes the facility and the severity

Header
: Includes the timestamp and the hostname or IP address of the machine

Message
: Includes the tag and the content

The *facility* describes the type of system that sends the log message. It allows log messages from different facilities to be handled differently. There are 24 facilities defined by the RFCs; here are a few:

- 0 (kernel)
- 1 (user-level)
- 2 (mail)
- 3 (system daemons)
- 4 (security/authorization messages)

The *severity* level is similar to the log level. Syslog defines eight different levels, with 0 being the highest and 7 being the lowest:

- 0 (Emergency)
- 1 (Alert)
- 2 (Critical)
- 3 (Error)
- 4 (Warning)
- 5 (Notice)
- 6 (Informational)
- 7 (Debug)

The timestamp and the hostname or IP address are self-explanatory. The *tag* is the name of the program that generated the message, while the *content* is the details of the log message.

Syslog is not implemented uniformly in different operating systems. A popular implementation of syslog is rsyslog, the default syslog implementation in many Linux variants including Debian, Ubuntu, openSUSE, and others.

Go provides a `log/syslog` package as part of the standard library to interface with syslog. However, it doesn't work on all systems. For example, it doesn't work with Windows because it's not implemented on Windows.

The example in this recipe is based on running against rsyslog on Ubuntu 20.04, and it should work on systems with rsyslog. However, I have not tried it on all systems and implementations.

Before we start on the Go code, you need to set up rsyslog to show the priority, header, and message parts. In rsyslog this is done using a template in the rsyslog configuration file.

Start by editing the */etc/rsyslog.conf* configuration file:

```
$ sudo vi /etc/rsyslog.conf
```

Add the template configuration after this line— `$ActionFileDefaultTemplate RSYSLOG_TraditionalFileFormat` in the configuration file:

```
$template gosyslogs,"%syslogseverity-text% %syslogfacility-text% %hostname%
%timegenerated% %syslogtag% %msg%\n"
$ActionFileDefaultTemplate gosyslogs
```

In this configuration, you name the template `gosyslogs`. You set it to show the severity first, followed by the facility, then the hostname and the timestamp, and finally, the tag and message.

Once you save this file, restart rsyslog:

```
sudo service rsyslog restart
```

Now that you have set up rsyslog, you can look at the code. Sending log messages to syslog using the `syslog` package is relatively straightforward:

```
var logger *log.Logger

func init() {
        var err error
        logger, err = syslog.NewLogger(syslog.LOG_USER|syslog.LOG_NOTICE, 0)
        if err != nil {
                log.Fatal("cannot write to syslog: ", err)
        }
}
```

```
func main() {
        logger.Print("hello syslog!")
}
```

You use the `NewLogger` function to create a logger, passing the syslog priority flags you want to set. The `syslog` package provides flags for the facility and the severity levels. You can *or* them together to send the facility code and the severity level. For the case of the preceding code, you send in `syslog.LOG_USER` indicating the *user* facility code, and `syslog.LOG_NOTICE` indicating the *notice* severity level.

Run the code first in the file named *main.go*:

```
$ go run main.go
```

Now check the syslogs. Run this on the command line:

```
$ sudo tail /var/log/syslog
```

You should see a bunch of log messages, but somewhere at the bottom, you should see something like this:

```
notice user myhostname Jan 26 15:30:08 /tmp/go-build2223050573/b001/exe/
main[163995]:
  hello syslog!
```

The first field is `notice`, the severity level, followed by `user`, which is the facility code. Next is `myhostname`, which is the hostname, followed by the timestamp.

The next field is the *tag*, which is the `/tmp/go-build2223050573/b001/exe/main[163995]` field in the log message. Why is it indicating that it's in the `/tmp` directory? That's because you're using `go run`. It will look different if you compile the code and run the binary file. The final field in the log message is the details of the log, which you print out using `logger`.

CHAPTER 5

Function Recipes

5.0 Introduction

Functions are sequences of code that are put together such that they can be activated (or called) as a single unit. Functions enable programmers to break up problems into smaller parts, making code easier to understand and reusable. There are many other names used for such a construct, including routine, subroutine, procedure, and of course, function. In Go, this construct is called function.

Functions are relatively simple to understand and use, and most of the time. However, certain function concepts, like anonymous functions and closures, are more advanced. This chapter will explain basic and some more advanced concepts in functions and how Go implements them.

5.1 Defining a Function

Problem

You want to define a function.

Solution

Define a function using the `func` keyword.

Discussion

A function needs to be defined before it can be called. Each function definition in Go starts with the `func` keyword. The function needs to have a name. As with any variable, if the name is capitalized, it is exported and visible outside of the package. Otherwise, it is visible only within the package. Each function can take in zero or

more *parameters*, which are inputs to the function. Each parameter must have a name and a data type, and they are placed after the name of the function within parentheses:

```
func myFunction(x int) {
    ...
}
```

In this example, `x` is the parameter's name, and its data type is `int`. The body of the function is within two curly brackets.

Functions can return zero or more values. The names of the return values are optional, but their data types are not. If the return value is named, it must be within parentheses, placed after the parameters:

```
func myfunction(x int) (y string) {
    ...
}
```

In the preceding example, `y` is the name of the return value, and its data type is `string`. You can also omit the name of the return value, in which case you can drop the parentheses:

```
func myfunction(x int) string {
    ...
}
```

Functions can be attached to structs. Such functions are called *methods*. In such cases you need to specify a *receiver* in the function definition:

```
type MyStruct struct {
    ...
}


func (s MyStruct) myfunction(x int) string {
    ...
}
```

In this example, `s` is the receiver for the function.

In all cases, the receiver, parameters, and return values (when named) are all accessible within the function.

Finally, if you want to allow multiple data types for the same function, you specify one or more *type parameters* with corresponding *type constraints* in the function declaration. Both type parameter and type constraint are defined within square brackets after the function name but before the parameters:

```
func myfunction[T int | float64] (x T) string {
    ...
}
```

In this example, `T` is the type parameter. As you can see, it is used in the function definition as the data type for the parameter (it can also be used in the return value). It can be used in the function body as well. The type constraint specifies which data types can be used in place of `T`. In this case, it can be an `int` or a `float64`.

5.2 Accepting Multiple Data Types with a Function

Problem

You want a function that can accept more than one data type.

Solution

Use generics to define a function that can take in more than a single function.

Discussion

Many data structures and algorithms can be used with different data types. For example, you can apply the same sorting algorithm on int, string, or float. When you implement an algorithm or write a function, you often need to specify the data type.

In Go, to create a function to add two numbers together, you need to know the data types of the two numbers before you can write the function:

```
func AddInt(a, b int) int {
        return a + b
}

func AddFloat(a, b float64) float64 {
        return a + b
}
```

As you can see, if you want to add two ints, you need to write one function, and if you're going to add two floats, you need to write another.

Generics is a programming language feature that allows you to write code for a generic data type instead of a specific one. This means you don't need to use a specific data type for your code. In the preceding case, generics allows you to create a single function that can add two ints or two floats.

Go implements generics using a mechanism called *type parameters*. Type parameters are abstract data types defined between the function name and the parameter list, within square brackets. Type constraints are requirements that type parameters must fulfill. Type constraints are special interfaces.

Using generics, you can define a single function `Add` that will add either ints or float64s:

```
func Add[T int | float64] (x T) T {
    a + b
}
```

In this example, the type parameter is `T`, and the type constraint is a union between `int` and `float64`.

The | operator allows you to create a union of types that creates a type constraint that allows both int and float64 types. Doing this kind of union is kind of tedious, so Go allows you to group them as an interface (which makes sense since type constraints are also interfaces):

```
type Number interface {
    int | float64
}
```

Go also provides a package under the experimental packages umbrella called `constraints` that provides some commonly used type constraint interfaces.

There are a number of type constraints, for example, the `Signed` constraint:

```
type Signed interface {
    ~int | ~int8 | ~int16 | ~int32 | ~int64
}
```

The tilde (~) operator indicates that this also applies to all custom types whose underlying type is that type. For example, `~int` indicates that this applies to int but also to any custom types whose underlying type is int.

An interesting constraint is the `Ordered` constraint, which looks like this:

```
type Ordered interface {
    Integer | Float | ~string
}
```

With this, you can also write the same code from the previous example like this:

```
import "golang.org/x/exp/constraints"

type Number interface {
    constraints.Integer | constraints.Float
}

func AddNumbers[T Number](a, b T) T {
    return a + b
}
```

5.3 Accepting a Variable Number of Parameters

Problem

You want to accept a variable number of parameters in a function.

Solution

Define a variadic function using `...` before the data type in the parameter definition.

Discussion

Sometimes you want a function to accept any number of parameters of the same type. You could, of course, use a slice, place all the data into the slice, and pass that slice over to the function. However, Go has a better mechanism—make it a variadic function. A *variadic* function is a function that allows any number of parameters:

```
func varFunc(str ...string) {
        for _, s := range str {
                fmt.Printf("%s ", s)
        }
        fmt.Println()
}
```

The `varFunc` function here is a variadic function as it has a parameter `str` that is variadic. It means you can pass zero or more strings into the function. For example, you can do this:

```
varFunc("the", "quick")
varFunc("the", "quick", "brown", "fox")
varFunc()
```

All three are valid, and this is what you should see:

```
the quick
the quick brown fox
```

The last function call has no parameters. This is permitted in variadic functions.

You can also have other parameters besides the variadic parameters:

```
func varFunc2(i int, str ...string) {
        for _, s := range str {
                fmt.Printf("%s ", s)
        }
        fmt.Println()
}
```

However, the variadic parameter must be the *last parameter* in the list. The variadic parameter passed into the function is converted into a slice, and you can manipulate it like a slice.

If you already have a slice and want to pass it to a variadic function, you can do this:

```
str := []string{"the", "quick", "brown", "fox"}
varFunc(str...)
```

The `...` after the `str` variable allows the slice to be passed into the variadic function.

5.4 Accepting Parameters of Any Type

Problem

You want to accept any type of data in a function.

Solution

Use the `any` type constraint or the empty interface `interface{}`.

Discussion

An interface in Go is a type that has a set of methods. Any type that implements the same methods is considered to be of that interface. Interfaces can be defined with the keyword `type ... interface`.

For example, this is an interface named `Stringer` from the `fmt` standard library:

```
type Stringer interface {
    String() string
}
```

Any type with a `String` method that returns a string is considered to have implemented the `Stringer` interface. It is how Go implements polymorphism.

Here's a trick question—what if the interface has no methods? An interface with no methods is the *empty interface* `interface{}`, and it is the set of all types. All types implement the empty interface, which means the empty interface represents all types. For convenience, the predeclared type `any` is an alias for the empty interface.

In this case, if a parameter is of the empty interface type, or `any`, it means you can pass in any data! Let's take a closer look by creating a simple function that accepts a single parameter with the `any` or `interface{}` type:

```
func anyFunc(a any) {
        fmt.Printf("value is %v\n", a)
}
```

Let's call the function with various data types:

```
anyFunc("hello world")
anyFunc(123)
anyFunc(123.456)
```

Running the preceding code will give you these results:

```
value is hello world
value is 123
value is 123.456
```

How about a struct?

```
type Dog struct {
        Name  string
        Age   int
        Breed string
}
snowy := Dog{"Snowy", 6, "Fox Terrier"}
anyFunc(snowy)
```

This works too!

```
value is {Snowy 6 Fox Terrier}
```

However, the irony is that because the parameter can be *anything*, you can't do anything with it (using `fmt.Printf` is kind of cheating). Why not? For example, if you want to add the parameter to another number, you need to know if the parameter is a number too, or at least something that can be converted into a number. In other words, you need to find out what type the parameter is.

In Go, the `reflect` package provides two ways of helping you to figure out what a variable is. The first is `reflect.TypeOf`, which tells you the variable type, and the second is `Kind`, which tells you what kind of variable it is. For primitive types, both are the same, but once you get a data structure, either built-in like `slice` or `map`, or a custom struct, these will be different. Here's how you can use the `reflect` package:

```
func anyFuncReflect(a any) {
        fmt.Printf("value is %v, type is %v, kind is %v\n", a, reflect.TypeOf(a),
                reflect.TypeOf(a).Kind())
}
```

If you pass different parameters to this function, you can see how they work:

```
anyFuncReflect("hello world")
anyFuncReflect(123)
anyFuncReflect(123.456)
anyFuncReflect(snowy)
anyFuncReflect([]int{1, 2, 3})
```

Here is the output:

```
value is hello world, type is string, kind is string
value is 123, type is int, kind is int
value is 123.456, type is float64, kind is float64
value is {Snowy 6 Fox Terrier}, type is functions.Dog, kind is struct
value is [1 2 3], type is []int, kind is slice
```

For the struct, the type is the struct name, while the kind is struct. For the slice of integers, the type is `[]int`, while the kind is slice.

Even though you know what it is now, you still can't use it. This is because from Go's perspective, the parameter is still `any`. To use the parameter `a`, you need to type-assert it. Type assertion, unlike type casting, doesn't convert an interface to another data type. It just provides access to the interface value's underlying value. In other words, type assertion tells the compiler what the underlying value is.

Here's how to do this in another function. In Go, you can use the comma, ok pattern on type assertion. This pattern returns a boolean (in a variable normally named `ok`) and the asserted type. If the assertion is fine, `ok` will be true; otherwise it will be false:

```
func anyFuncAssert(a any) {
        dog, ok := a.(Dog)
        if ok {
                fmt.Printf("Name is %s, age is %d, breed is %s\n", dog.Name,
                dog.Age,
                        dog.Breed)
        } else {
                fmt.Println("Not a dog")
        }
}
```

Once you have type-asserted `a` using `a.(Dog)`, you can use the variable as a `Dog` struct.

What if you want only certain data types to be accepted but not others? In this case, you can use the Go generics implementation—the type parameter mechanism:

```
func anyFuncGeneric[T int | float64](a T) {
        fmt.Printf("value is %v, type is %v, kind is %v\n", a, reflect.TypeOf(a),
                reflect.TypeOf(a).Kind())
}
```

Rerun the following code:

```
anyFuncGeneric("hello world")
anyFuncGeneric(123)
anyFuncGeneric(123.456)
snowy := Dog{"Snowy", 6, "Fox Terrier"}
anyFuncGeneric(snowy)
```

You will get errors. The compiler will not allow you to proceed further:

```
string does not implement int|float64
Dog does not implement int|float64
```

5.5 Creating an Anonymous Function

Problem

You want to create a function that is not attached to any identifier.

Solution

Define a function with no name as a function value or attach it to a variable.

Discussion

Anonymous functions are simply functions that do not have a name. They are useful when you want to create a function within a function.

You can define an anonymous function like this:

```
func(a, b int) (c int) {
        return a + b
}
```

As you can see, there is no name in the function definition. This creates a function literal, which you can assign to a variable. If you inspect the variable, you'll see that its type is `func(int, int) int`, and its kind is `func`:

```
func anonFunc1() {
        anon := func(a, b int) (c int) {
                return a + b
        }
        fmt.Println("type is:", reflect.TypeOf(anon), "\nkind is:",
                reflect.TypeOf(anon).Kind())
        fmt.Println(anon(1, 2))
}
```

You can call the anonymous function through the variable by passing it the parameters like any normal function. You can even call the function literal directly by placing the parameters after the function literal itself:

```
func anonFunc2() {
        anon := func(a, b int) (c int) {
                return a + b
        }(1, 2)
        fmt.Println(anon)
}
```

Both the `anonFunc1` and `anonFunc2` functions produce the same results. The `anon` variable in `anonFunc1` contains a `func` while the `anon` variable in `anonFunc2` contains the integer 3.

As mentioned earlier, anon is of type func(int, int) int, and if you were wondering if this means you can declare a variable with this type, then you are correct. In fact, you can create a type out of it:

```
func anonFunc3() {
        type myfunc func(int, int) int
        var anon myfunc
        anon = func(a, b int) (c int) {
                return a + b
        }
        fmt.Println(anon(1, 2))
}
```

In this code example, you create a type named myfunc, which is of the signature func(int, int) int. You can use this type to declare the anon variable, which can be called subsequently by passing it parameters, just as before. When you call anonFunc3, it will print 3 as the output.

You can also create a function that accepts a parameter of type myfunc. You can then call this function and pass it a function literal:

```
type myfunc func(int, int) int

func caller() {
        anonFunc4(func(a, b int) (c int) {
                return a + b
        })
}

func anonFunc4(f myfunc) {
        fmt.Println(f(1, 2))
}
```

In the preceding code, anonFunc4 accepts a parameter of type myfunc, which is a function. When you call anonFunc4 in the caller you can pass it a function literal. This function literal will be executed, and the results will be returned to anonFunc4 as a parameter to fmt.Println.

5.6 Creating a Function That Maintains State After It Is Called

Problem

You want to create a function that will retain its state even after it finishes execution.

Solution

Create a Go closure by returning a function from a function call.

Discussion

A closure is a function with an associated environment that persists outside the scope of the function body. This is implemented by returning a function from a function call, for example:

```
func outerFunc() func() int {
        count := 0
        return func() int {
                count++
                return count
        }
}
```

In this example, `outerFunc` is a function that returns an anonymous function of type `func() int`. This anonymous function can access the variable `count` and anything else that is within the scope of `outerFunc`.

This is what happens when you call the `outerFunc` function:

```
next := outerFunc()
fmt.Println(next())
fmt.Println(next())
fmt.Println(next())
```

When you call the `outerFunc` function, it returns a function assigned to the `next` variable. Subsequently, `next` is called repeatedly. This is the output from running the code:

```
1
2
3
```

The variable `count` within the scope of `outerFunc` is persisted along with the anonymous function returned by `outerFunc` and stored in `next`. As a result, `count` also persists across every subsequent call, and the number increments.

The `outerFunc` function is a *closure* because it is *closed* with the environment in `outerFunc`. This allows the environment to persist even though `outerFunc` has completed execution and its scope is gone.

Why do you need closures? They have many uses, but writing middleware for web applications is one of the most well-known.

Web application middleware are functions that can be added to the web application's request/response pipeline to handle common tasks like logging or access control. In Go, it's common to implement web application middleware using closures.

To see how closure is being used, here's a simple web application that shows "Hello World" when you access the web application at `/hello`:

```
package main

import (
        "fmt"
        "net/http"
)

func main() {
        http.HandleFunc("/hello", hello)
        http.ListenAndServe(":8000", nil)
}

func hello(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello World")
}
```

The web application is simple. It has a handler named `hello`, which is used as a parameter in the `HandleFunc` function to set up a handler for the web application.

If you want to log to screen or file each time a handler is called, and you also want to know how long the handler takes to execute its tasks, you could create a small function that allows you to log and call that function in each handler. It's tedious, but it works. Logging the time, however, is even messier because you need to insert code at the beginning and run some deferred code to get the end timing to figure out the execution time.

Alternatively, you can create a piece of middleware that will do both:

```
package main

import (
        "fmt"
        "log"
        "net/http"
        "reflect"
        "runtime"
        "time"
)

func main() {
        http.HandleFunc("/hello", logger(hello))
        http.ListenAndServe(":8000", nil)
}

func hello(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello World")
}

func logger(f http.HandlerFunc) http.HandlerFunc {
        return func(w http.ResponseWriter, r *http.Request) {
                start := time.Now()
                f(w, r)
```

```
                end := time.Now()
                name := runtime.FuncForPC(reflect.ValueOf(f).Pointer()).Name()
                log.Printf("%s (%v)", name, end.Sub(start))
        }
}
```

You create a `logger` function that accepts a function of type `HandlerFunc` (which is simply of type `func(ResponseWriter, *Request)`) and returns a function of type `HandlerFunc` as well. Inside this function, you return a closure that starts a timer, calls the function that was passed in as a parameter, ends the timer, and logs the name of the function, as well as the time it took to execute, to the screen.

Then when you set up the handlers, you simply wrap the `hello` handler with the `logger` closure middleware. When you run the web application and call the URL `/hello`, you will see this on the terminal:

```
% go run main.go
2022/06/15 00:27:23 main.hello (97.083µs)
```

In Recipe 5.5, you learned about anonymous functions, which are functions without a name. A closure can be an anonymous function, but it doesn't always have to be one. In the previous example, the anonymous function in `logger` is a closure and not `logger` itself. In the earlier example, `outerFunc` is a closure but not an anonymous function.

CHAPTER 6

String Recipes

6.0 Introduction

String manipulation is one of the most common tasks in any programming language. Most programs deal with text in one way or another, whether directly interacting with users or communicating between machines. Text is probably the closest thing we have to a universal medium, and string as data is everywhere. Being able to manipulate strings is a critical capability in your arsenal as a programmer.

Go has several packages used for string manipulation. The `strconv` package focuses on converting to or from strings. The `fmt` package provides functions to format strings using verbs as replacements, much like in C. The `unicode/utf8` and `unicode/utf16` packages have functions used in Unicode-encoded strings. The `strings` package has functions to do many of the string manipulations we see so if you're not sure what you need, that's the most likely location to look.

6.1 Creating Strings

Problem

You want to create strings.

Solution

Use either the double quotes (`""`) or the backtick (or backquote) (``` `` ```) to create string literals. Use the single quotes (`''`) to create character literals.

Discussion

In Go, `string` is a read-only (immutable) slice of bytes. It can be any byte and doesn't need to be in any encoding or format. This is unlike other programming languages, where strings are sequences of characters. In Go, a character can be represented by more than a single byte. This is in line with the Unicode standard, which defines a code point to represent a value within a codespace. A character, in this case, can be represented by more than a single code point. In Go, code points are also called *runes*, and a rune is an alias for the type `int32`, just as a byte is an alias for the type `uint8`, which represents an unsigned 8-bit integer.

As a result if you index a string, you will end up with a byte and not a character. In any case Go doesn't have a character data type—bytes and runes are used instead. A byte represents ASCII characters and runes represent Unicode characters in UTF-8 encoding. To be clear, it doesn't mean that there are no characters in Go, just that there is no `char` data type, just `byte` and `rune`.

Characters in Go are created using single quotes:

```
var c = 'A'
```

In this case, the data type of the variable `c` is `int32` or a rune, by default. If you want it to be a byte, you can explicitly specify the type:

```
var c byte = 'A'
```

Strings in Go can be created using double quotes or backticks:

```
var str = "A simple string"
```

Strings created using double quotes can have escape characters in them. For example, a very common escape character is the newline, represented by a backslash followed by an *n*: `\n`:

```
var str = "A simple string\n"
```

Another common use of escape characters is to escape a double quote itself so it can be used within a string created with a double quote:

```
var str = `A "simple" string`
```

Strings created using backticks are considered "raw" strings. Raw strings ignore all formatting, including escape characters. In fact, you can create a multiline string using backticks. For example, this is not possible using double quotes—it'll be a syntax error:

```
var str = "
A
simple
string
"
```

However, if you replace the double quotes with backticks, `str` will be a multiline string:

```
var str = `
A
simple
string
`
```

This is because whatever comes in between the backticks is not processed by the Go compiler (it is "raw").

6.2 Converting String to Bytes and Bytes to String

Problem

You want to convert string to bytes and bytes to string.

Solution

Typecast a string to an array of bytes using `[]byte(str)` and typecast an array of bytes to a string using `string(bytes)`.

Discussion

Strings are slices of bytes, so you can convert a string to an array of bytes directly through typecasting:

```
str := "This is a simple string"
bytes := []byte(str)
```

Converting an array of bytes to a string is also done through typecasting:

```
bytes := []byte{84, 104, 105, 115, 32, 105, 115, 32, 97, 32, 115, 105, 109, 112,
                                  108, 101, 32, 115, 116, 114, 105, 110, 103}
str := string(bytes)
```

6.3 Creating Strings from Other Strings and Data

Problem

You want to create a string from other strings or data.

Solution

There are various ways of doing this, including direct concatenation and `strings.Join`, using `fmt.Sprint` and using `strings.Builder`.

Discussion

At times you want to create strings from other strings or data. One rather straightforward way of doing this is to concatenate strings and other data:

```
var str string = "The time is " + time.Now().Format(time.Kitchen) + " now."
```

The `Now` function returns the current time, formatted by the `Format` method and returned as a string. When you concatenate the strings, you will get this:

```
The time is 5:28PM now.
```

Another way of doing this is to use the `Join` function in the `string` package:

```
var str string = strings.Join([]string{"The time is",
                    time.Now().Format(time.Kitchen),
                    "now."}, " ")
```

This is straightforward as well because the function takes in an array of strings and, given the separator, puts them together.

So far, both ways shown are about putting strings together. Obviously, you can convert different data types into strings before joining them, but sometimes you just want to Go to do it. For this, you have the `fmt.Sprint` function and its various variants. Let's look at the simplest and most direct variant:

```
var str string = fmt.Sprint("The time is ", time.Now().Format(time.Kitchen),
        " now.")
```

This doesn't seem very different from the `Join` or the direct concatenation because all three parameters are strings. Actually, `fmt.Sprint` and its variants take parameters of type `any`, which means it can take in any data type. In other words, you can pass in the `Time` struct instance that's returned by `Now` directly:

```
var str string = fmt.Sprint("The time is ", time.Now(), " now.")
```

A popular variant of `fmt.Sprint` is the formatted variant, that is, `fmt.Sprintf`. Using this variant is slightly different—the first parameter is the format string, where you can place different verb formats at different locations within the string. The parameters after the first are the data values that can be replaced with the verbs:

```
var str string = fmt.Sprintf("The time is %v now.", time.Now())
```

There is no associated verb for a `Time` struct, so you use the `%v`, which will format the value in the default format.

Finally, the `string` package also provides another way of creating strings using the `strings.Builder` struct. Using `Builder` to create strings is a bit more involved, as it requires you to add the data piece by piece. Let's take a look at using `Builder`:

```
var builder strings.Builder
builder.WriteString("The time is ")
builder.WriteString(time.Now().Format(time.Kitchen))
builder.WriteString(" now.")
var str string = builder.String()
```

The idea is simple. You create a `Builder` struct, then write data to it bit by bit before finally extracting the final string using the `String` method. The `Builder` struct has a few other methods, including `Write`, which takes in an array of bytes; `WriteByte`, which takes in a single byte; and `WriteRune`, which takes in a single rune. However, as you can see, they are all strings. How about other data types? Do you need to convert all other data types to string, byte, or rune first? No, because `Builder` is a `Writer` (it implements a `Write` method), you can actually use another way of writing different data types into it:

```
var builder strings.Builder
fmt.Fprint(&builder, "The time is ")
fmt.Fprint(&builder, time.Now())
fmt.Fprint(&builder, " now.")
var str string = builder.String()
```

Here you're using `fmt.Fprint` to write whatever data type you want into the `builder` and extract the final string using `String`.

You've seen quite a few ways of putting a string together using different pieces of data, both string and other types of data. Some are very straightforward (just add them together), and others are more deliberate. But which is the best way of doing it? Here's a look at the performance of these various ways:

```
package string

import (
        "fmt"
        "strings"
        "testing"
        "time"
)

func BenchmarkStringConcat(b *testing.B) {
        for i := 0; i < b.N; i++ {
                _ = "The time is " + time.Now().Format(time.Kitchen) + " now."
        }
}

func BenchmarkStringJoin(b *testing.B) {
        for i := 0; i < b.N; i++ {
                _ = strings.Join([]string{"The time is", time.Now().Format
                (time.Kitchen),
            "now."}, " ")
        }
}
```

```
func BenchmarkStringSprint(b *testing.B) {
        for i := 0; i < b.N; i++ {
                _ = fmt.Sprint("The time is ", time.Now().Format(time.Kitchen),
                               " now.")
        }
}

func BenchmarkStringSprintDiff(b *testing.B) {
        for i := 0; i < b.N; i++ {
                _ = fmt.Sprint("The time is ", time.Now(), " now.")
        }
}

func BenchmarkStringSprintf(b *testing.B) {
        for i := 0; i < b.N; i++ {
        _ = fmt.Sprintf("The time is %v now.", time.Now().Format(time.Kitchen))
        }
}

func BenchmarkStringSprintfDiff(b *testing.B) {
        for i := 0; i < b.N; i++ {
                _ = fmt.Sprintf("The time is %s now.", time.Now())
        }
}

func BenchmarkStringBuilderFprint(b *testing.B) {
        for i := 0; i < b.N; i++ {
                var builder strings.Builder
                fmt.Fprint(&builder, "The time is ")
                fmt.Fprint(&builder, time.Now().Format(time.Kitchen))
                fmt.Fprint(&builder, " now.")
                _ = builder.String()
        }
}

func BenchmarkStringBuilderWriteString(b *testing.B) {
        for i := 0; i < b.N; i++ {
                var builder strings.Builder
                builder.WriteString("The time is ")
                builder.WriteString(time.Now().Format(time.Kitchen))
                builder.WriteString(" now.")
                _ = builder.String()
        }
}
```

Now run the benchmark from the command line:

```
$ % go test -bench=BenchmarkString -benchmem
```

These are the results:

```
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch06_string
BenchmarkStringConcat-10                        5787976           206.7 ns/op
BenchmarkStringJoin-10                          5121637           235.0 ns/op
BenchmarkStringSprint-10                        3680838           323.8 ns/op
BenchmarkStringSprintDiff-10                    1541514           779.9 ns/op
BenchmarkStringSprintf-10                       4032438           297.8 ns/op
BenchmarkStringSprintfDiff-10                   1610212           740.9 ns/op
BenchmarkStringBuilderFprint-10                 2580783           464.2 ns/op
BenchmarkStringBuilderWriteString-10            4866556           247.0 ns/op
PASS
ok      github.com/sausheong/gocookbook/ch06_string     13.025s
```

It might (or might not) come as a surprise that the simplest way is the most performant. Using `fmt.Sprint` and anything that uses `any` (or the empty interface `interface{}`) is simply less efficient.

6.4 Converting Strings to Numbers

Problem

You want to convert strings to numbers.

Solution

Use the `Atoi` or the `Parse` functions in the `strconv` package to do string conversions. Use functions to convert strings to numbers, and use `Itoa` or the `Format` functions to convert numbers to strings.

Discussion

The `strconv` package is true to its name and is used mainly for strings conversion. Broadly speaking, there are two sets of functions in the `strconv` package that do string conversions. The `Parse` functions convert strings to numbers, and the `Format` functions convert numbers to strings. If you're unsure which ones to use, remember this—parsing reads strings whereas formatting creates strings.

Parsing strings into numbers seems to be limited in usage, but it can be handy when dealing with formatted text data, for example, JSON or YAML, or even XML. Formatted text data is popular because it is human-readable, but the drawback is that everything ends up being a string. Parsing strings into something more directly usable, such as numbers, becomes really useful.

Start with something simple. You want to parse a string that shows an integer and produces an actual integer:

```
i, err := strconv.Atoi("123") // equivalent to ParseInt("123", 10, 0)
```

The `strconv` package provides a convenience function to convert a string to an integer. This is quite easy to remember because `Atoi` literally converts *(a)lphanumeric* to *(i)nteger*.

The equivalent of `Atoi` using the `Parse` functions is `ParseInt(s, 10, 0)`, where `s` is the string representing the number.

The `ParseInt` function, as the name suggests, parses a string into an integer. You can specify the base (0, 2 to 36) and the bit size (0 to 64). You can use `ParseInt` for signed or unsigned integers—just place a + or – in front of the number:

```
i, err := strconv.ParseInt("123", 10, 0)
```

The bit size parameter specifies the integer type the result must fit into. Bit sizes 0, 8, 16, 32, and 64 correspond to int, int8, int16, int32, and int64, respectively. Note that the bit size here refers to the bit size of input number (which is in the form of a string). The returned output, `i` in the case of the preceding example, is always `int64`.

Similarly, the `ParseFloat` function parses a string into a float. The bit size parameter specifies the precision, 32 for `float32`, 64 for `float64`, etc., and as with `ParseInt`, the bit size parameter refers to the input number and the returned output; in this case, `f` is always a `float64`:

```
f, err := strconv.ParseFloat("1.234", 64)
```

`ParseBool` can be useful when you're trying to parse a string that represents a boolean value. It accepts `1`, `t`, `T`, `TRUE`, `true`, `True`, `0`, `f`, `F`, `FALSE`, `false`, and `False`:

```
b, err := strconv.ParseBool("TRUE")
```

In this code, `b` is a boolean with the value `true`.

All the `Parse` functions return `NumError`, including `Atoi`. `NumError` provides additional information about the error, including the function that was called, the number passed in, and why it failed:

```
str := "Not a number"
_, err := strconv.Atoi(str)
if err != nil {
    e := err.(*strconv.NumError)
    fmt.Println("Func:", e.Func)
    fmt.Println("Num:", e.Num)
    fmt.Println("Err:", e.Err)
    fmt.Println(err)
}
```

This is what you see if you run this code:

```
Func: Atoi
Num: Not a number
Err: invalid syntax
strconv.Atoi: parsing "Not a number": invalid syntax
```

6.5 Converting Numbers to Strings

Problem

You want to convert numbers to strings.

Solution

Use the `Itoa` or the `Format` functions in the `strconv` package to convert numbers to strings.

Discussion

We discussed the `strconv` package and the `Parse` functions in the previous recipe. In this recipe, we'll talk about the `Format` functions and how you can use them to convert numbers to strings.

Formatting numbers into strings is the reverse of parsing strings into numbers. In cases where data needs to be communicated through text formats, formatting numbers can be useful. One frequent usage of formatting numbers into strings is when you need to show more readable numbers to users. For example, instead of showing 1.66666666 to the user, you would want to show 1.67. This is also commonly used when displaying currency.

Just as parsing strings has `Atoi`, formatting strings has `Itoa`. As the name suggests, it's the reverse of `Atoi`—it converts an integer into a string:

```
str := strconv.Itoa(123) // equivalent to FormatInt(int64(123), 10)
```

Notice that `Itoa` doesn't return an error. In fact, none of the `Format` functions return errors. It makes sense—it is always possible to make a number a string, while it's not the case in the reverse.

As before, `Itoa` is a convenient function for `FormatInt`. However, you must first ensure the input number parameter is always an `int64`. `FormatInt` also requires a `base` parameter where `base` is an integer between 2 and 36, both numbers included. This means `FormatInt` can potentially convert binary numbers to string:

```
str := strconv.FormatInt(int64(123), 10)
```

The code returns a string `"123"`. What if you specify a base of 2?

```
str := strconv.FormatInt(int64(123), 2)
```

This will return a string `"1111011"`. This means that you can use `FormatInt` to convert a number in one base to another, at least to display it as a string.

The `FormatFloat` function is a bit more complicated than `FormatInt`. It converts a floating-point number to a string according to a given format and precision. The formats available in `FormatFloat` for decimal numbers (base 10) are:

- `f` (no exponent)
- `e` and `E` (with exponent)
- `g` and `G` - `e` or `E` respectively (if the exponent is large, it will follow the `e` or `E` format; else it will be without exponent, like `f`)

The other formats are `b` for binary numbers (base 2) and `x` and `X` for hexadecimal numbers.

The precision describes the number of digits (excluding the exponents) to be printed out. A precision of value –1 allows Go to select the smallest number of digits such that `ParseFloat` returns the entire number.

Some code will make things clearer:

```
var v float64 = 123456.123456
var s string

s = strconv.FormatFloat(v, 'f', -1, 64)
fmt.Println("f (prec=-1)\t:", s)
s = strconv.FormatFloat(v, 'f', 4, 64)
fmt.Println("f (prec=4)\t:", s)
s = strconv.FormatFloat(v, 'f', 9, 64)
fmt.Println("f (prec=9)\t:", s)
```

You're using a float value with 64-bit precision, `float64`, and you will compare the precision of –1 with the precision of 4. You also use the same format, *f*. When you run the code, you should see the following output:

```
f (prec=-1)     : 123456.123456
f (prec=4)      : 123456.1235
f (prec=9)      : 123456.123456000
```

Now let's try with the *e* format and the same set of precisions:

```
s = strconv.FormatFloat(v, 'e', -1, 64)
fmt.Println("\ne (prec=-1)\t:", s)
s = strconv.FormatFloat(v, 'E', -1, 64)
fmt.Println("E (prec=-1)\t:", s)
s = strconv.FormatFloat(v, 'e', 4, 64)
```

```
fmt.Println("e (prec=4)\t:", s)
s = strconv.FormatFloat(v, 'e', 9, 64)
fmt.Println("e (prec=9)\t:", s)
```

The following output is what you should see:

```
e (prec=-1)     : 1.23456123456e+05
E (prec=-1)     : 1.23456123456E+05
e (prec=4)      : 1.2346e+05
e (prec=9)      : 1.234561235e+05
```

In case you didn't realize, both lowercase *e* and uppercase *E* are exactly the same, except the exponent letter *e* is lower- or uppercase. Finally, let's look at the *g* format:

```
s = strconv.FormatFloat(v, 'g', -1, 64)
fmt.Println("\ng (prec=-1)\t:", s)
s = strconv.FormatFloat(v, 'G', -1, 64)
fmt.Println("G (prec=-1)\t:", s)
s = strconv.FormatFloat(v, 'g', 4, 64)
fmt.Println("g (prec=4)\t:", s)
```

The following is the output:

```
g (prec=-1)     : 123456.123456
G (prec=-1)     : 123456.123456
g (prec=4)      : 1.235e+05
```

6.6 Replacing Multiple Characters in a String

Problem

You want to replace parts in a string with another string.

Solution

You can use the `strings.Replace` function or `strings.ReplaceAll` function to replace the selected string. You can also use the `strings.Replacer` type to create replacers.

Discussion

The easiest way to replace parts of a string with another string is to use the `strings.Replace` function.

The `strings.Replace` function is quite straightforward. Just pass it a string, the old string you want to replace, and the new string you want to replace it with. Here's the code, using this quote from *Great Expectations* by Charles Dickens:

```
var quote string = `I loved her against reason, against promise,
against peace, against hope, against happiness,
against all discouragement that could be.`
```

Run a few replacements using `Replace`:

```
replaced := strings.Replace(quote, "against", "with", 1)
fmt.Println(replaced)
replaced2 := strings.Replace(quote, "against", "with", 2)
fmt.Println("\n", replaced2)
replacedAll := strings.Replace(quote, "against", "with", -1)
fmt.Println("\n", replacedAll)
```

The last parameter tells `Replace` the number of matches to replace. If the last parameter is –1, `Replace` will match every instance:

```
I loved her with reason, against promise,
against peace, against hope, against happiness,
against all discouragement that could be.

 I loved her with reason, with promise,
against peace, against hope, against happiness,
against all discouragement that could be.

 I loved her with reason, with promise,
with peace, with hope, with happiness,
with all discouragement that could be.
```

There is also a `ReplaceAll` function, which is more of a convenience function that calls `Replace` with the last parameter set to –1.

The `Replacer` type in the `strings` package allows you to make multiple replacements all at the same time. This is a lot more convenient if you need to do a lot of replacements:

```
replacer := strings.NewReplacer("her", "him", "against", "for", "all", "some")
replaced := replacer.Replace(quote)
fmt.Println(replaced)
```

You just need to provide a list of replacement strings as the parameters. In the previous code, you replaced "her" with "him," "against" with "for," and "all" with "some." All the replacements are done at the same time. If you run the code, you will get the following results:

```
I loved him for reason, for promise,
for peace, for hope, for happiness,
for some discouragement that could be.
```

So is it better to use `Replace` or create a `Replacer`? It takes another line of code to create the replacer, obviously. But how about its performance? Start with replacing just one word:

```
func BenchmarkOneReplace(b *testing.B) {
        for i := 0; i < b.N; i++ {
                strings.Replace(quote, "her", "him", 1)
        }
}
```

```
func BenchmarkOneReplacer(b *testing.B) {
        replacer := strings.NewReplacer("her", "him")
        b.ResetTimer()
        for i := 0; i < b.N; i++ {
                replacer.Replace(quote)
        }
}
```

If it's just one string to replace, `Replace` is faster. There is just more overhead for a simple replacement:

```
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch06_string
BenchmarkOneReplace-10           7264310               156.9 ns/op
BenchmarkOneReplacer-10          4336489               276.0 ns/op
PASS
ok      github.com/sausheong/gocookbook/ch06_string     3.151s
```

To do multiple replacements:

```
func BenchmarkReplace(b *testing.B) {
        for i := 0; i < b.N; i++ {
                strings.Replace(quote, "against", "with", -1)
        }
}

func BenchmarkReplacerCreate(b *testing.B) {
        for i := 0; i < b.N; i++ {
                strings.NewReplacer("against", "with")
        }
}

func BenchmarkReplacer(b *testing.B) {
        replacer := strings.NewReplacer("against", "with")
        b.ResetTimer()
        for i := 0; i < b.N; i++ {
                replacer.Replace(quote)
        }
}
```

You don't need to create a `Replacer` each time because you can use the replacer multiple times if you want to do the same replacements for different strings. At the same time, if you have a lot of replacements, it's easier to use a replacer:

```
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch06_string
BenchmarkReplace-10              2250291               532.1 ns/op
BenchmarkReplacerCreate-10      31878366                37.13 ns/op
BenchmarkReplacer-10             4671319               255.0 ns/op
```

```
PASS
ok      github.com/sausheong/gocookbook/ch06_string     4.547s
```

As you can see from the results, with more strings to replace, using `Replacer` becomes more efficient.

6.7 Creating a Substring from a String

Problem

You want to create a substring from a string.

Solution

Treat a string like an array or a slice and take a substring out of the string.

Discussion

In Go, a string is a slice of bytes. As a result, if you want to take a substring out of a string, you can do what you would do with any slice. Take the quote from the previous recipes (*Great Expectations* by Charles Dickens):

```
var quote string = `I loved her against reason, against promise,
against peace, against hope, against happiness,
against all discouragement that could be.`
```

If you want to extract the words "against reason" from the quote, you can do this:

```
quote[12:26]
```

This is simple enough, but how do you know the position of the words without manually counting the letters in the quote? Like in many programming languages, you just need to find the index of the substring:

```
strings.Index(quote, "against reason")
```

The `strings.Index` function gives you the index of the first substring that matches the second parameter. In this case, the index is 12. This will give you the starting position of the substring. To find the end position of the substring, add the length of the substring to the index:

```
i := strings.Index(quote, "against reason")
j := i + len("against reason")
fmt.Println(quote[i:j])
```

A word of caution—when slicing a string this way, don't count the number of characters manually. Not all encodings have a single character represented by a single byte. Always use `index` and `len` to find the position and length of the string to slice.

6.8 Checking if a String Contains Another String

Problem

You want to check if a string contains another string.

Solution

Use the `Contains` functions in the `strings` package. If the string you want to check is a suffix or a prefix, you can use the `HasSuffix` or the `HasPrefix` functions.

Discussion

Checking if a string has a substring is quite easy in Go. You can use the `strings.Contains` function and pass in both the string and substring, and it will return true or false accordingly. We'll use the quote from *Great Expectations* by Charles Dickens again:

```
var quote string = `I loved her against reason, against promise,
against peace, against hope, against happiness,
against all discouragement that could be.`
```

The `Contains` function checks if the quote contains the string "against":

```
var has bool = strings.Contains(quote, "against")
```

Alternatively, you can use `strings.Index`, and if the returned result is < 0 it means the substring is not found in the string. The performance is the same in either function, not surprisingly since `Contains` is just a convenience function around `Index`. Another alternative is to use the `Count` function, which returns the number of times the substring is found in the string, but this is usually a poorer alternative (unless you need to know the count anyway) because the performance is worse than either.

If you want to find out if the substring is the prefix of the string, you can use the `HasPrefix` function:

```
strings.HasPrefix(quote, "I loved")
```

Of course, you can directly slice the length of the prefix from the string and check it yourself:

```
prefix := "I loved"
if quote[:len(prefix)] == prefix {
    ... // do whatever you want if the string has the prefix
}
```

You can do the same for suffixes as well with the `HasSuffix` function:

```
strings.HasSuffix(quote, "could be.")
```

You can also directly slice the string for the suffix and compare it:

```
suffix := "could be."
if quote[len(quote)-len(suffix):] != suffix {
    ... // do whatever you want if the string has the prefix
}
```

6.9 Splitting a String Into an Array of Strings or Combining an Array of Strings Into a String

Problem

You want to create an array of strings by splitting up a string or create a string by combining an array of strings.

Solution

Use the `Split` functions in the `strings` package to split up a string and the `Join` function to combine the array of strings into a single string.

Discussion

Many functions take in an array of strings. You might want to tackle words in a string instead of individual bytes. You might be dealing with data delimited by a separator, like in a delimited text format like CSV or TSV. Whichever case it may be, quickly splitting up a string into an array of strings is useful.

In Go, you can do this using the `strings.Split` function. You can use the quote from *Great Expectations* by Charles Dickens again:

```
var quote string = `I loved her against reason, against promise,
against peace, against hope, against happiness,
against all discouragement that could be.`
```

The `Split` function splits a string into an array of strings, given the separator:

```
array := strings.Split(quote, " ")
fmt.Printf("%q", array)
```

The preceding code uses a space as the separator, so this is what you will get:

```
["I" "loved" "her" "against" "reason," "against" "promise," "\nagainst" "peace,"
"against" "hope," "against" "happiness," "\nagainst" "all" "discouragement"
"that" "could" "be."]
```

You might notice that some elements in the array have newline characters and punctuation marks because the original string has them. It's probably not what you want. Or worse, if you have multiple spaces, your array will look pretty messy with a lot of additional empty string elements. Of course, you can clean it up by brute force

later on, but there are simpler ways. Let's start with removing newline characters, and leading and trailing spaces.

The `strings` package has a function called `Fields` that can split a string considering one or more consecutive spaces as defined by `uniform.isSpace`.

You can swap out `Split` and replace it with the `Fields` function:

```
array := strings.Fields(quote)
fmt.Printf("%q", array)
```

You should see this:

```
["I" "loved" "her" "against" "reason," "against" "promise," "against" "peace,"
"against" "hope," "against" "happiness," "against" "all" "discouragement" "that"
"could" "be."]
```

The newline characters are gone, but the punctuation marks are still around. Removing the punctuation marks (in this case, the commas and the full stops within the elements) is a bit more complicated. You need to use the `FieldsFunc` function and pass in a function that will determine whether it should be part of the separator:

```
f := func(c rune) bool {
    return unicode.IsPunct(c) || !unicode.IsLetter(c)
}
array := strings.FieldsFunc(quote, f)
fmt.Printf("%q", array)
```

In this code, you create a function `f` that considers consecutive punctuations and nonletters as part of the separator. Then you pass this function into `FieldsFunc` for it to be executed against the string. You should see the following results:

```
["I" "loved" "her" "against" "reason" "against" "promise" "against" "peace"
"against" "hope" "against" "happiness" "against" "all" "discouragement" "that"
"could" "be"]
```

As you can see, you have also removed the commas and the full stop. The `FieldsFunc` function is very versatile; I've given only a very simple example. If you work a lot with splitting strings, this will be a powerful function you can use to do many things.

What if you want to split the string just for the first nine elements and put the rest in a single string? The `SplitN` function does exactly that, with `n` being the number of elements to have in the resulting array—in this case, it's 10:

```
array := strings.SplitN(quote, " ", 10)
fmt.Printf("%q", array)
```

You will see that there are 10 elements in the resulting array:

```
["I" "loved" "her" "against" "reason," "against" "promise," "\nagainst" "peace,"
"against hope, against happiness, \nagainst all discouragement that could be."]
```

Sometimes you want to keep the delimiter after you split the string. Go has a function called `SplitAfter` that does this:

```
array := strings.SplitAfter(quote, " ")
fmt.Printf("%q", array)
```

When you use `SplitAfter`, each element ends with a space (the delimiter) except the final element:

```
["I " "loved " "her " "against " "reason, " "against " "promise, " "\nagainst "
"peace, " "against " "hope, " "against " "happiness, " "\nagainst " "all "
"discouragement " "that " "could " "be."]
```

6.10 Trimming Strings

Problem

You want to remove the leading and trailing characters of a string.

Solution

Use the `Trim` functions in the `strings` package.

Discussion

When processing strings, it's quite common to encounter trailing or leading whitespaces or other unnecessary characters. You often want to remove these characters before storing or processing the strings further. String trimming removes characters from the start or the end of the string, but not within the string.

In Go, there are several `Trim` functions in the `strings` package that can help you with trimming strings.

Let's start with the `Trim` function. It takes in a string and a *cutset*, which is a string consisting of one or more Unicode code points, and returns a string with all leading and trailing code points removed:

```
var str string = ", and that is all."
var cutset string = ",. "
trimmed := strings.Trim(str, cutset) // "and that is all"
```

In this code, you want to remove the leading comma, whitespace, and the trailing full stop. To do this, you use a cutset consisting of these three Unicode code points, so the cutset string ends up being `",. "` (comma, full stop, and space, respectively).

The `Trim` function removes both trailing and leading characters. If you want to remove trailing characters only, you can use the `TrimRight` function, or if you want to remove the leading characters only, you can use the `TrimLeft` function:

```
trimmedLeft := strings.TrimLeft(str, cutset)   // "and that is all."
trimmedRight := strings.TrimRight(str, cutset) // ", and that is all"
```

The earlier `Trim` functions remove any characters in the cutset. However, if you want to remove an entire leading substring (or a prefix) you can use the `TrimPrefix` function:

```
trimmedPrefix := strings.TrimPrefix(str, ", and ")      // "that is all."
```

Similarly, if you want to remove an entire trailing substring (or a suffix) you can use the `TrimSuffix` function:

```
trimmedSuffix := strings.TrimSuffix(str, " all.") // ", and that is"
```

The `Trim` functions allow you to remove any leading or trailing characters or string. However, the most commonly removed characters are usually whitespaces, which can be newlines (`\n`) or tabs (`\t`), or carriage returns (`\r`). For convenience, Go provides a `TrimSpace` function that simply removes trailing and leading whitespaces:

```
trimmed := strings.TrimSpace("\r\n\t Hello World \t\n\r") // Hello World
```

The last set of `Trim` functions are `TrimFunc`, `TrimLeftFunc`, and `TrimRightFunc`. As the names indicate, these allow you to substitute the cutset string with a function that will inspect the leading or trailing or both Unicode code points and ensure they satisfy the conditions:

```
f := func(c rune) bool {
        return unicode.IsPunct(c) || !unicode.IsLetter(c)
}
trimmed := strings.TrimFunc(str, f) // "and that is all"
```

These `TrimFunc` functions allow you finer control over string trimming, which can be useful if you have unexpected rules for removing the leading or trailing characters.

6.11 Capturing String Input from the Command Line

Problem

You want to capture user input string data from the command line.

Solution

Use the `Scan` functions in the `fmt` package to read a single string from standard input. To read a string separated by spaces, use `ReadString` on a `Reader` wrapped around `os.Stdin`.

Discussion

If your Go program runs from the command line, you might need to get string input from the user. This is where the `Scan` function from the `fmt` package comes in handy.

You can use `Scan` to get input from the user by creating a variable and then passing a reference to that variable into `Scan`:

```
package main

import "fmt"

func main() {
        var input string
        fmt.Print("Please enter a word: ")
        n, err := fmt.Scan(&input)
        if err != nil {
                fmt.Println("error with user input:", err, n)
        } else {
                fmt.Println("You entered:", input)
        }
}
```

If you run this code, the program will wait for your input at `fmt.Scan` and will continue only when you enter an input. Once you have entered some data, `Scan` will store the data in the `input` variable.

This is what you should see:

```
% go run scan.go
Please enter a word: Hello
You entered: Hello
```

The documentation does not explicitly mention that you need to pass in a reference to a variable. You can pass in a variable by value, and it will compile. However, if you do that you will get an error:

```
n, err := fmt.Scan(input)
if err != nil {
        fmt.Println("error with user input:", err, n)
}
```

If you run the preceding code, you will get this:

```
% go run scan.go
Please enter a word: error with user input: type not a pointer: string 0
```

The `Scan` function can take in more than one parameter, and each parameter represents a user input separated by a space. Try this again with two inputs:

```
func main() {
        var input1, input2 string
        fmt.Print("Please enter two words: ")
```

```
        n, err := fmt.Scan(&input1, &input2)
        if err != nil {
                fmt.Println("error with user input:", err, n)
        } else {
                fmt.Println("You entered:", input1, "and", input2)
        }
}
```

If you run this and enter the words *Hello* and *World*, they will be captured and stored into `input1` and `input2`, respectively. You should see the following output:

```
% go run scan.go
Please enter two words: Hello World
You entered: Hello and World
```

This seems a bit limited. What if you want to capture a string that has spaces in it? For example, you want to get a user to input a sentence. In that case, you can use the `ReadString` function on a `Reader` wrapped around `os.Stdin`:

```
func main() {
        reader := bufio.NewReader(os.Stdin)
        fmt.Print("Please enter many words: ")
        input, err := reader.ReadString('\n')
        if err != nil {
                fmt.Println("error with user input:", err)
        } else {
                fmt.Println("You entered:", input)
        }
}
```

If you run the code, you should see this:

```
% go run scan.go
Please enter many words: Many words here and still more to go
You entered: Many words here and still more to go
```

You should know that `Scan` can be used to get more than just string input from users. It can also be used to get numbers and so on.

6.12 Escaping and Unescaping HTML Strings

Problem

You want to escape or unescape HTML strings.

Solution

Use the `EscapeString` and `UnescapeString` functions in the `html` package to escape or unescape HTML strings.

Discussion

HTML is a text-based markup language that structures a web page and its content. It is usually interpreted by a browser and displayed. Much of HTML is described within HTML tags. For example, `<a>` is an anchor tag and `<img>` is an image tag. Similarly, other characters like `&` and `"` have specific meanings in HTML.

But what if you want to show those characters in HTML itself? For example, the ampersand (`&`) is a commonly used character. The less than and greater than (`<` and `>`) characters are also commonly used. If you don't intend these symbols to have any meaning in HTML, you need to convert them into HTML character entities. For example:

- < (less than) becomes <
- > (greater than) becomes >
- & (ampersand) becomes &

This applies to all such character entities. The process of converting HTML characters into entities is called *HTML escaping*, and the reverse is called *HTML unescaping*.

Go has a pair of functions called `EscapeString` and `UnescapeString` in the `html` package that can be used to escape or unescape HTML:

```
str := "<b>Rock & Roll</b>"
escaped := html.EscapeString(str) // "&lt;b&gt;Rock &amp; Roll&lt;/b&gt;"
```

Unescaping reverts the escaped HTML to the original string:

```
unescaped := html.UnescapeString(escaped) // "<b>Rock & Roll</b>"
```

You might notice that there is also an `HTMLEscapeString` function in the `html/template` package. The results of both functions are the same.

6.13 Using Regular Expressions

Problem

You want to use regular expressions to do string manipulation.

Solution

Use the `regex` package and parse the regular expression using the `Compile` function to return a `Regexp` struct. Then use the `Find` functions to match the pattern and return the string.

Discussion

Regular expressions are a popular notation for describing a search pattern in a string. When a particular string is in the set described by a regular expression, the regular expression matches the string. Regular expressions are available in many languages.

Go has an entire standard package dedicated to regular expressions called `regex`. The syntax of the regular expressions is the same general syntax used by Perl, Python, and other languages.

Using the `regex` package is quite straightforward. First, create a `Regexp` struct instance from the regular expressions. With this struct, you can call any number of `Find` functions that will return the strings or the index of the strings that match.

While it might seem there are a lot of `Find` functions in the `regex` package, mostly attached as methods to the `Regexp` struct, there is a general pattern to them:

- The ones without `All` will return only the first match.
- The ones with `All` potentially return all the matches in the string, depending on the `n` parameter.
- The ones with `String` will return strings or slices of strings.
- The ones without `String` will return as an array of bytes, `[]byte`.
- The ones with `Index` return the index of the match.

In the following code snippets, we will use the quote from *Great Expectations* by Charles Dickens as with the other recipes in this chapter:

```
var quote string = `I loved her against reason, against promise,
against peace, against hope, against happiness,
against all discouragement that could be.`
```

Start with creating a `Regexp` struct instance that you will use later:

```
re, err := regexp.Compile(`against [\w]+`)
```

The regular expression here is `against [\w]+`.

You use backticks to create the regular expression string because regular expressions use a lot of backslashes, and these would be interpreted differently if you use double quotes. The regular expression you use matches against a pattern within a string that starts with *against* and has a word after it.

A convenient alternative to `Compile` is the `MustCompile` function. This function does exactly the same thing as `Compile`, except that it doesn't return an error. Instead, if the regular expression doesn't compile, the function will panic.

Once you have the regular expression set up, you can use it to find matches. As mentioned earlier, there are many `Find` methods, but this recipe will cover only a few. One of the most straightforward methods is `MatchString`, which tells you if the regular expression has any matches in the string:

```
re.MatchString(quote) // true
```

At times, besides checking if the regular expression has any matches, you also want to return the matching string. You can use `FindString` to do this:

```
str := re.FindString(quote) // "against reason"
```

Here you find a string from the `quote` string, using the regular expression you set up earlier. It returns the first match, so the returned string is `against reason`. If you want to return all matches, you have to use a method with `All` in it, like `FindAllString`:

```
strs := re.FindAllString(quote, -1)
fmt.Println(strs)
```

The second parameter in `FindAllString`, like all `All` methods, indicates the number of matches you want returned. If you want to return all matches, you need to use a negative number, in this case, `-1`. The returned values are an array of strings.

If you run the code, which also prints out the array of strings, this is what you get:

```
[against reason against promise against peace against hope against happiness
against all]
```

Besides returning the matched strings, sometimes you want to find the locations of the matches. In this case, you can use the `Index` functions; for example, the `FindStringIndex`:

```
locs := re.FindStringIndex(quote) // [12 26]
```

This returns a two-element slice of integers, which indicates the position of the first match. If you use these two integers on the `quote` itself, you will be able to extract the matching substring:

```
quote[locs[0]:locs[1]] // against reason
```

As before, to get all matches, you need to use the `All` method, so in this case, you can use the `FindAllStringIndex` method:

```
allLocs := re.FindAllStringIndex(quote, -1)
fmt.Println(allLocs)
```

This will return a two-dimensional slice of all matches:

```
[[12 26] [28 43] [46 59] [61 73] [75 92] [95 106]]
```

In addition to finding and indexing regular expressions, you can replace the matched strings altogether using `ReplaceAllString`. This is a simple example:

```
replaced := re.ReplaceAllString(quote, "anything")
fmt.Println(replaced)
```

If you run the preceding code, this is what you should see:

```
I loved her anything, anything,
anything, anything, anything,
anything discouragement that could be.
```

Beyond a simple replacement, you can replace the matched string with the output of a function that takes in the matched string and produces another string. Here's a quick example:

```
replaced = re.ReplaceAllStringFunc(quote, strings.ToUpper)
fmt.Println(replaced)
```

You are replacing all the matched strings with the uppercase version of the string by using the `strings.ToUpper` function. This is the result of running the code:

```
I loved her AGAINST REASON, AGAINST PROMISE,
AGAINST PEACE, AGAINST HOPE, AGAINST HAPPINESS,
AGAINST ALL discouragement that could be.
```

Instead of making both words in the matched string uppercase, what if you want only the second word to be uppercase? You can create a simple function to do this:

```
f := func(in string) string {
        split := strings.Split(in, " ")
        split[1] = strings.ToUpper(split[1])
        return strings.Join(split, " ")
}
replaced = re.ReplaceAllStringFunc(quote, f)
fmt.Println(replaced)
```

If you run this code, you will get this:

```
I loved her against REASON, against PROMISE,
against PEACE, against HOPE, against HAPPINESS,
against ALL discouragement that could be.
```

Regular expressions are very flexible and used in many places. In Go, they can be used for very powerful string manipulation. However, there is a word of caution. The `regex` package in Go supports the regular expression syntax accepted by RE2. This guarantees the regular expressions to run in time linear to the size of the input. As a result, some syntax like `lookahead` and `lookbehind` are not supported. If you're more familiar with the syntax supported by PCRE library, you might want to check and make sure your regular expressions work the way you expect.

CHAPTER 7

General Input/Output Recipes

7.0 Introduction

Input and output (more popularly known as I/O) are how a computer communicates with the external world. Typical input into a computer refers to the keystrokes from a keyboard or clicks from or movement of a mouse. It can also refer to other external input sources like a camera or a microphone, gaming joystick, and so on. Output often refers to whatever is shown on the screen (or on the terminal) or printed out on a printer. I/O can also refer to network connections and to files. I/O is a key part of developing software and most programming languages, including Go, have standard libraries that can read from input and write to output.

This chapter explores some common Go recipes for managing I/O. We'll warm up with some basic I/O recipes, then talk about files in general. In the next few chapters, we'll move on to CSV, followed by JSON and binary files.

The `io` package is the base package for input and output in Go and contains interfaces for I/O and a few convenient functions. The main and the most commonly used interfaces are `Reader` and `Writer`, but there are several variants of these, like `ReadWriter`, `TeeReader`, `WriterTo`, and many more.

Generally, these interfaces are nothing more than a descriptor for functions. For example, a struct that is a `Reader` has a `Read` function. A struct that is a `WriterTo` has a `WriteTo` function. Some interfaces combine two or more interfaces. For example, the `ReadWriter` combines the `Reader` and `Writer` interfaces and has both the `Read` and `Write` functions.

The recipes in this chapter explain more about how these interfaces are used.

7.1 Reading from an Input

Problem

You want to read from an input.

Solution

Use the `io.Reader` interface to read from an input.

Discussion

Go uses the `io.Reader` interface to represent the ability to read from an input stream of data. Many packages in the Go standard library and third-party packages use the `Reader` interface to allow data to be read:

```
type Reader interface {
        Read(p []byte) (n int, err error)
}
```

Any struct that implements the `Read` function is a `Reader`. Let's say you have a reader (a struct that implements the `Reader` interface). To read data from the reader, you make a slice of bytes, and you pass that slice to the `Read` method:

```
bytes = make([]byte, 1024)
reader.Read(bytes)
```

It might look counterintuitive and seem like you would want to read data from `bytes` into the reader, but you're actually reading the data from the reader into `bytes`. Think of it as the data flowing from left to right, from the reader into `bytes`.

`Read` will fill the slice of bytes only to its capacity. If you want to read everything from the reader, you can use the `io.ReadAll` function:

```
bytes, err := io.ReadAll(reader)
```

This looks more intuitive because the `ReadAll` reads from the reader passed into the parameter and returns the data into `bytes`. In this case, the data flows from the reader on the right into the bytes on the left.

You will often find functions that expect a reader as an input parameter. Let's say you have a string, and you want to pass the string to the function. What can you do? You can create a reader from the string using the `strings.NewReader` function, then pass it into the function:

```
str := "My String Data"
reader := strings.NewReader(str)
```

You can now pass `reader` into functions that expect a reader.

7.2 Writing to an Output

Problem

You want to write to an output.

Solution

Use the `io.Writer` interface to write to an output.

Discussion

The interface `io.Writer` looks like `io.Reader`:

```
type Writer interface {
        Write(p []byte) (n int, err error)
}
```

When you call `Write` on an `io.Writer`, you are writing the bytes to the underlying data stream:

```
bytes = []byte("Hello World")
writer.Write(bytes)
```

You might notice that this is the reverse of `io.Reader` in Recipe 7.1 even though the method signature looks very similar. In `Reader`, you call the `Read` method to read from the struct into the `bytes` variable, whereas here, you call the `Write` method to write from the `bytes` variable into the struct. In this case, the data flows from right to left, from `bytes` into the writer.

A common pattern in Go is for a function to take in a writer as a parameter. The function calls the `Write` function on the writer, and later you can extract the data from the writer, for example:

```
var buf bytes.Buffer
fmt.Fprintf(&buf, "Hello %s", "World")
s := buf.String() // s == "Hello World"
```

The `bytes.Buffer` struct is a `Writer` (it implements the `Write` function), so you can easily create one and pass it to the `fmt.Fprintf` function, which takes in an `io.Writer` as its first parameter. The `fmt.Fprintf` function writes data onto the buffer, and you can extract the data from it later.

This pattern of using a writer to pass data around by writing to it, then extracting it later is quite common in Go. An example of this in the standard library is in the HTTP handlers with the `http.ResponseWriter`.

The following is a handler function named `myHandler`:

```
func myHandler(w http.ResponseWriter, r *http.Request) {
    w.Write([]bytes("Hello World"))
}
```

In this handler, you write data, a slice of bytes representing the string "Hello World," to the `ResponseWriter`. The data is stored within the `ResponseWriter` implementation and passed around for further processing until it is finally sent to the browser.

7.3 Copying from a Reader to a Writer

Problem

You want to copy from a reader to a writer.

Solution

Use the `io.Copy` function to copy from a reader to a writer.

Discussion

Sometimes you read from a reader because you want to write it to a writer. The process can take a few steps to read everything from a reader into a buffer and then write it to the writer again. Instead of doing this, you can use the `io.Copy` function. The `io.Copy` function takes from a reader and writes to a writer all in one function.

Here is an example of using `io.Copy`.

A common way to download a file is to use `http.Get` to get a reader, which you read, and then you use `os.WriteFile` to write to a file:

```
// using a random 1MB test file
var url string = "http://speedtest.ftp.otenet.gr/files/test1Mb.db"

func readWrite() {
        r, err := http.Get(url)
        if err != nil {
                log.Println("Cannot get from URL", err)
        }
        defer r.Body.Close()
        data, _ := io.ReadAll(r.Body)
        os.WriteFile("rw.data", data, 0755)
}
```

When you use `http.Get` to download a file, you get an `http.Response` struct, `r`. The content of the file is in the `Body` variable of the `http.Response` struct, which is an `io.ReadCloser`. A `ReadCloser` is an interface that groups a `Reader` and a `Closer` so you can treat it just like a reader. You use the `io.ReadAll` function to read the data from `Body` and then `os.WriteFile` to write it to a file.

That's simple enough, but what about the performance of the function. You use the benchmarking capabilities that are part of the standard Go tools. First, you create a test file:

```
import "testing"

func BenchmarkReadWrite(b *testing.B) {
        readWrite()
}
```

In this test file, instead of a function that starts with `Testxxx` you create a function that starts with `Benchmarkxxx`, which takes in a parameter `b` that is a reference to `testing.B`.

The benchmark function is very simple. You just call your `readWrite` function. Run it from the command line and see how it performs:

```
$ go test -bench=. -benchmem
```

You use the `-bench=.` flag telling Go to run all the benchmark tests and `-benchmem` flag to show memory benchmarks. You should see the following output:

```
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch07_io
BenchmarkReadWrite
BenchmarkReadWrite-10 1 1916690833 ns/op        5271952 B/op    218 allocs/op
PASS
ok      github.com/sausheong/gocookbook/ch07_io 2.051s
```

You ran a benchmark test for a function that downloaded a 1 MB file. The test only ran one time, and it took 1.91 seconds. It also took 5.27 MB of memory and 218 distinct memory allocations.

As you can see, it's quite an expensive operation to download a 1 MB file. After all, it takes about 5 MB of memory to download a 1 MB file. Alternatively, you can use `io.Copy` to do pretty much the same thing for a lot less memory:

```
func copy() {
        r, err := http.Get(url)
        if err != nil {
                log.Println("Cannot get from URL", err)
        }
        defer r.Body.Close()
        file, _ := os.Create("copy.data")
```

```
        defer file.Close()
        writer := bufio.NewWriter(file)
        io.Copy(writer, r.Body)
        writer.Flush()
}
```

First, you create a file for the data, here using `os.Create`. Next, you create a buffered writer using `bufio.NewWriter`, wrapping around the file. This will be used in the `Copy` function, copying the contents of the response `Body` into the buffered writer. Finally, you flush the writer's buffers and make the underlying writer write to the file.

If you run this `copy` function it works the same way, but how does the performance compare? Go back to our benchmark and add another benchmark function for this `copy` function:

```
import "testing"

func BenchmarkReadWrite(b *testing.B) {
        readWrite()
}

func BenchmarkCopy(b *testing.B) {
        copy()
}
```

Run the benchmark again and this is what you should see:

```
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch07_io
BenchmarkReadWrite
BenchmarkReadWrite-10 1 3379608041 ns/op        5271872 B/op    218 allocs/op
BenchmarkCopy
BenchmarkCopy-10      1 1618772042 ns/op          43200 B/op     62 allocs/op
PASS
ok      github.com/sausheong/gocookbook/ch07_io 5.409s
```

This time the `readWrite` function took 3.37 seconds, used 5.27 MB of memory, and did 218 memory allocations. The `copy` function, however, only took 1.61 seconds, used 43.2 kB of memory, and did 62 memory allocations.

The `copy` function is about twice as fast and uses only a fraction (less than 1%) of the memory. With really large files, if you're using the `io.ReadAll` and `os.WriteFile`, you might run out of memory quickly.

7.4 Reading from a Text File

Problem

You want to read a text file into memory.

Solution

You can use the `os.Open` function to open the file, followed by `Read` on the file. Alternatively, you can use the simpler `os.ReadFile` function to do it in a single function call.

Discussion

Reading and writing to the filesystem are basic things a programming language needs to do. Of course, you can always store data in memory but, sooner or later, if you need to persist the data beyond a shutdown, you need to store it somewhere. There are several ways that data can be persisted, but persisting it to the local filesystem is probably the most common.

Read everything in one go

The easiest way to read a text file is to use `os.ReadFile`. Say you want to read from a text file named *data.txt* that has the following content:

```
hello world!
```

To read the file, just give the name of the file as a parameter to `os.ReadFile` and you're done!

```
data, err := os.ReadFile("data.txt")
if err != nil {
  log.Println("Cannot read file:", err)
}
fmt.Println(string(data))
```

This will print out `hello world!`.

Opening a file and reading from it

Reading a file by opening it and then doing a read on it is more flexible but takes a few more steps. First, you need to open the file:

```
// open the file
file, err := os.Open("data.txt")
if err != nil {
  log.Println("Cannot open file:", err)
}
// close the file when we are done with it
defer file.Close()
```

You can do this using `os.Open`, which returns a `File` struct instance that is read-only. If you want to open it in different modes, you can use `os.OpenFile`. It's good practice to set up the file for closing using the `defer` keyword, which will close the file just before the function returns.

Next, you need to create a byte array to store the data:

```
// get some info from the file
stat, err := file.Stat()
if err != nil {
  log.Println("Cannot read file stats:", err)
}
// create the byte array to store the read data
data := make([]byte, stat.Size())
```

To do this, you need to know how large the byte array should be, and that should be the size of the file. You use the `Stat` method on the file to get a `FileInfo`. Then you call the `Size` method on `FileInfo` to get the size of the file.

Once you have the byte array, you can pass it as a parameter to the `Read` method on the file struct:

```
// read the file
bytes, err := file.Read(data)
if err != nil {
  log.Println("Cannot read file:", err)
}
fmt.Printf("Read %d bytes from file\n", bytes)
fmt.Println(string(data))
```

This will store the read data into the byte array and return the number of bytes read. If all goes well you should see something like this from the output:

```
Read 13 bytes from file
Hello World!
```

There are a few more steps, but you have the flexibility of reading parts of the whole document, and you can also do other stuff in between opening the file and reading it.

7.5 Writing to a Text File

Problem

You want to write data to a text file.

Solution

You can use the `os.Open` function to open the file, followed by `Write` on the file. Alternatively, you can use the `os.WriteFile` function to do it in a single function call.

Discussion

Just as in reading a file, there are a couple of ways to write to a file.

Writing to a file in one go

Given the data, you can write to a file in one go using `os.WriteFile`:

```
data := []byte("Hello World!\n")

err := os.WriteFile("data.txt", data, 0644)
if err != nil {
  log.Println("Cannot write to file:", err)
}
```

The first parameter is the name of the file, the data is in a byte array, and the final parameter is the Unix file permissions you want to give to the file. If the file doesn't exist, this will create a new file. If it exists, it will remove all the data in the file and write the new data into it but without changing the permissions.

Creating a file and writing to it

Writing to a file by creating the file and then writing to it is a bit more involved but it's also more flexible. First, you need to create or open a file using the `os.Create` function:

```
data := []byte("Hello World!\n")
// write to file and read from file using the File struct
file, err := os.Create("data.txt")
if err != nil {
  log.Println("Cannot create file:", err)
}
defer file.Close()
```

This will create a new file with the given name and mode `0666` if the file doesn't exist. If the file exists, this will remove all the data in it. As before, you would want to set up the file to be closed at the end of the function, using the `defer` keyword.

Once you have the file you can write to it directly using the `Write` method and pass the byte array to it:

```
bytes, err := file.Write(data)
if err != nil {
  log.Println("Cannot write to file:", err)
}
fmt.Printf("Wrote %d bytes to file\n", bytes)
```

This will return the number of bytes that were written to the file. As before, while it takes a few more steps, breaking up the steps between creating a file and writing to it gives you more flexibility to write in smaller chunks instead of everything at once.

7.6 Using a Temporary File

Problem

You want to create a temporary file for use.

Solution

Use the `os.CreateTemp` function to create a temporary file, and then remove it once you don't need it anymore.

Discussion

A temporary file is a file that's created to store data temporarily while the program is doing something. It's meant to be deleted or copied to permanent storage once the task is done. In Go, you can use the `os.CreateTemp` function to create a temporary file. Then afterwards, you can remove it.

Different operating systems store their temporary files in different places. Regardless of where it is, Go will let you know where it is using the `os.TempDir` function:

```
fmt.Println(os.TempDir())
```

You need to know because the temp files created by `os.CreateTemp` will be created there. Normally you wouldn't care, but because you're trying to analyze step by step how the temp file gets created, you want to know exactly where it is. When you execute this statement, you should see something like this:

```
/var/folders/nj/2xd4ssp94zz41gnvsyvth38m0000gn/T/
```

This is the directory that your computer tells Go (and some other programs) to use as a temporary directory. You can use this directory directly, or you can create a subdirectory here using the `os.MkdirTemp` function:

```
tmpdir, err := os.MkdirTemp(os.TempDir(), "mytmpdir_*")
if err != nil {
        log.Println("Cannot create temp directory:", err)
}
defer os.RemoveAll(tmpdir)
```

The first parameter to `os.MkdirTemp` is the temporary directory, and the second parameter is a pattern string. The function will apply a random string to replace the `*` in the pattern string. It is also a good practice to defer the cleaning up of the temporary directory by removing it using `os.RemoveAll`.

Next, you're creating the actual temporary file using `os.CreateTemp`, passing it the temporary directory you just created and also a pattern string for the filename, which works the same as the temporary directory:

```
tmpfile, err := os.CreateTemp(tmpdir, "mytmp_*")
if err != nil {
        log.Println("Cannot create temp file:", err)
}
```

With that, you have a file and everything else works the same way as any other file:

```
data := []byte("Some random stuff for the temporary file")
_, err = tmpfile.Write(data)
if err != nil {
        log.Println("Cannot write to temp file:", err)
}
err = tmpfile.Close()
if err != nil {
        log.Println("Cannot close temp file:", err)
}
```

If you didn't choose to put your temporary files into a separate directory (which you delete and also everything in it when you're done), you can use `os.Remove` with the temporary file name like this:

```
defer os.Remove(tmpfile.Name())
```

CHAPTER 8

CSV Recipes

8.0 Introduction

The CSV (comma-separated values) format is a file format in which tabular data (numbers and text) can be easily written and read in a text editor. CSV is widely supported, and most spreadsheet programs, such as Microsoft Excel and Apple Numbers, support CSV. Consequently, many programming languages, including Go, come with libraries that produce and consume the data in CSV files.

It might surprise you that CSV has been around for more than 50 years. The IBM Fortran compiler supported it in OS/360 back in 1972. If you're not quite sure what that is, OS/360 is the batch processing operating system developed by IBM for its System/360 mainframe computer. So yes, one of the first uses for CSV was for Fortran in an IBM mainframe computer.

CSV is not very standardized, and not all CSV formats are separated by commas, either. Sometimes it can be a tab or a semicolon or other delimiters. However, there is an RFC specification for CSV—RFC 4180, though not everyone follows that standard.

The Go standard library has an `encoding/csv` package that supports RFC 4180 and helps you read and write CSV.

8.1 Reading the Whole CSV File

Problem

You want to read a CSV file into memory for use.

Solution

Use the `encoding/csv` package and `csv.ReadAll` to read all data in the CSV file into a 2D array of strings.

Discussion

Say you have a file like this:

```
id,first_name,last_name,email
1,Sausheong,Chang,sausheong@email.com
2,John,Doe,john@email.com
```

The first row is the header, the next two rows are data for the user. Here's the code to open the file and read it into the 2D array of strings:

```
file, err := os.Open("users.csv")
if err != nil {
 log.Println("Cannot open CSV file:", err)
}
defer file.Close()
reader := csv.NewReader(file)
rows, err := reader.ReadAll()
if err != nil {
 log.Println("Cannot read CSV file:", err)
}
```

First, you open the file using `os.Open`. This creates an `os.File` struct instance (which is an `io.Reader`) that you can use as a parameter to `csv.NewReader`. The `csv.NewReader` creates a new `csv.Reader` struct instance that can be used to read data from the CSV file. With this CSV reader, you can use `ReadAll` to read all the data in the file and return a 2D array of strings `[][]string`.

You might be surprised that this is a 2D array of strings. What if the CSV row item is an integer? Or a boolean or any other type? Remember that CSV files are text files, so there is no way for you to differentiate if a value is anything other than a string. In other words, all values are assumed to be string, and if you think otherwise you need to cast it to something else.

8.2 Reading a CSV File One Row at a Time

Problem

You want to read a CSV file one record at a time.

Solution

Use the `encoding/csv` package and `csv.Read`.

Discussion

Reading a CSV file all at once is good, but if you have a very large CSV file it might be easier to read it one row at a time. You can use the same CSV file as in Recipe 8.1, with the following content:

```
id,first_name,last_name,email
1,Sausheong,Chang,sausheong@email.com
2,John,Doe,john@email.com
```

The first row is the header, and the next two rows are data for the user. To read the file row by row, use `csv.NewReader` to create a new `csv.Reader` struct:

```
file, err := os.Open("users.csv")
if err != nil {
 log.Println("Cannot open CSV file:", err)
}
defer file.Close()
reader := csv.NewReader(file)
for {
    record, err := reader.Read()
    if err == io.EOF {
        break
    }
    if err != nil {
        log.Println("Cannot read CSV file:", err)
    }
    for value := range record {
        fmt.Printf("%s\n", record[value])
    }
}
```

As in Recipe 8.1, you open the file using `os.Open`. This creates an `io.Reader` that you can use as a parameter to `csv.NewReader`. The `csv.NewReader` creates a new `csv.Reader` struct. With this CSV reader, you read each record one at a time until `io.EOF` is encountered. The record returned by `Read` is a slice of strings, and each string is the value of a column within a CSV row.

8.3 Unmarshalling CSV Data Into Structs

Problem

You want to unmarshal CSV data into structs instead of a 2D array of strings.

Solution

Read the CSV into a 2D array of strings then store it into structs.

Discussion

For some other formats like JSON or XML, it's common to unmarshal the data read from files (or anywhere) into structs. You can also do this in CSV, though you need to do a bit more work.

Say you want to put the data into a `User` struct:

```
type User struct {
   Id         int
   firstName  string
   lastName   string
   email      string
}
```

If you want to unmarshal the data in the 2D array of strings to the `User` struct, you need to convert each item yourself:

```
var users []User
for _, row := range rows {
 id, _ := strconv.ParseInt(row[0], 0, 0)
 user := User{Id: int(id),
   firstName: row[1],
   lastName:  row[2],
   email:     row[3],
 }
 users = append(users, user)
}
```

In this example, because the user ID is an integer, you would use `strconv.ParseInt` to convert the string into an integer before using it to create the `User` struct.

At the end of the `for` loop you will have an array of `User` structs. If you print that out, this is what you should see:

```
{0  first_name  last_name  email}
{1  Sausheong  Chang  sausheong@email.com}
{2  John  Doe  john@email.com}
```

8.4 Removing the Header Line

Problem

If your CSV file has a line of headers that are column labels, you will get that as well in your returned 2D array of strings or array of structs. You want to remove it.

Solution

Read the first line using `Read` and then continue reading the rest.

Discussion

When you use `Read` on the reader, you will read the first line and then move the cursor to the next line. If you use `ReadAll` afterward, you can read the rest of the file into the rows that you want:

```
file, err := os.Open("users.csv")
if err != nil {
 log.Println("Cannot open CSV file:", err)
}
defer file.Close()
reader := csv.NewReader(file)
reader.Read() // use Read to remove the first line
rows, err := reader.ReadAll()
if err != nil {
 log.Println("Cannot read CSV file:", err)
}
```

This will give you something like this:

```
{1  Sausheong  Chang  sausheong@email.com}
{2  John  Doe  john@email.com}
```

8.5 Using Different Delimiters

Problem

CSV doesn't necessarily use commas as delimiters. You want to read a CSV file that has a delimiter that is not a comma.

Solution

Set the `Comma` variable in the `csv.Reader` struct instance to the delimiter used in the file and read as before.

Discussion

Perhaps the file you want to read has semicolons as delimiters:

```
id;first_name;last_name;email
1;Sausheong;Chang;sausheong@email.com
2;John;Doe;john@email.com
```

You need to to set the `Comma` in the `csv.Reader` struct instance you created earlier to a semicolon:

```
file, err := os.Open("users2.csv")
if err != nil {
  log.Println("Cannot open CSV file:", err)
}
defer file.Close()
reader := csv.NewReader(file)
reader.Comma = ';' // change Comma to the delimiter in the file
rows, err := reader.ReadAll()
if err != nil {
  log.Println("Cannot read CSV file:", err)
}
```

8.6 Ignoring Rows

Problem

You want to ignore certain rows when reading the CSV file.

Solution

Use comments in the file to indicate the rows to be ignored. Then enable coding in the `csv.Reader` and read the file as before.

Discussion

If you want to ignore certain rows, what you'd like to do is simply comment those rows out. Well, in CSV you can't because comments are not in the standard. However, with the Go `encoding/csv` package, you can specify a comment rune that, if you place at the beginning of the row, ignores the entire row.

Say you have this CSV file:

```
id,first_name,last_name,email
1,Sausheong,Chang,sausheong@email.com
# 2,John,Doe,john@email.com
```

To enable commenting, set the `Comment` variable in the `csv.Reader` struct instance that you got from `csv.NewReader`:

```
file, err := os.Open("users.csv")
if err != nil {
  log.Println("Cannot open CSV file:", err)
}
defer file.Close()
reader := csv.NewReader(file)
reader.Comment = '#' // lines that start with this will be ignored
rows, err := reader.ReadAll()
if err != nil {
  log.Println("Cannot read CSV file:", err)
}
```

When you run this, you'll see:

```
{0 first_name last_name email}
{1 Sausheong Chang sausheong@email.com}
```

8.7 Writing CSV Files

Problem

You want to write data from memory into a CSV file.

Solution

Use the `encoding/csv` package and `csv.Writer` to write to file.

Discussion

You had fun reading CSV files; now you have to write one. Writing is quite similar to reading. First, you need to create a file (an `io.Writer`):

```
file, err := os.Create("new_users.csv")
if err != nil {
  log.Println("Cannot create CSV file:", err)
}
defer file.Close()
```

The data to write to the file needs to be in a 2D array of strings. Remember, if you don't have the data as a string, convert it into a string before you do this. Create a `csv.Writer` struct instance with the file. After that you can call `WriteAll` on the writer and the file will be created. This writes all the data in your 2D string array into the file:

```
data := [][]string{
 {"id", "first_name", "last_name", "email"},
 {"1", "Sausheong", "Chang", "sausheong@email.com"},
 {"2", "John", "Doe", "john@email.com"},
}
writer := csv.NewWriter(file)
err = writer.WriteAll(data)
if err != nil {
 log.Println("Cannot write to CSV file:", err)
}
```

8.8 Writing to File One Row at a Time

Problem

Instead of writing everything in your 2D string, you want to write to the file one row at a time.

Solution

Use the `Write` method on `csv.Writer` to write a single row.

Discussion

Writing to file one row at a time is pretty much the same, except you will want to iterate the 2D array of strings to get each row and then call `Write`, passing that row. You will also need to call `Flush` whenever you want to write the buffered data to the `Writer` (the file). In the following example code you called `Flush` after you had written all the data to the writer, but that's because you don't have a lot of data. If you have a lot of rows, you would probably want to flush the data to the file once in a while. To check if there are any problems with writing or flushing, you can call `Error`:

```
writer := csv.NewWriter(file)
for _, row := range data {
 err = writer.Write(row)
 if err != nil {
  log.Println("Cannot write to CSV file:", err)
 }
}
writer.Flush()
```

CHAPTER 9

JSON Recipes

9.0 Introduction

JavaScript Object Notation (JSON) is a lightweight data-interchange text format. It's meant to be read by humans but also easily read by machines and is based on a subset of JavaScript. JSON was originally defined by Douglas Crockford but is currently described by RFC 7159, as well as ECMA-404. JSON is used in REST-based web services, although they don't necessarily need to accept or return JSON data.

JSON is popular with RESTful web services but it's also frequently used for configuration. Creating and consuming JSON is commonplace in many web applications, from getting data from a web service to authenticating your web application through a third-party authentication service to controlling other services.

Go supports JSON in the standard library using the `encoding/json` package.

9.1 Parsing JSON Data Byte Arrays to Structs

Problem

You want to read JSON data byte arrays and store them into structs.

Solution

Create structs to contain the JSON data and then use `Unmarshal` in the `encoding/json` package to unmarshal the data into the structs.

Discussion

Parsing JSON with the `encoding/json` package is straightforward:

1. Create structs to contain the JSON data.
2. Unmarshal the JSON string into the structs.

Here's a sample JSON file, containing data on the *Star Wars* character, Luke Skywalker, taken from SWAPI, the Star Wars API. The data has been taken and stored in a file named *skywalker.json*:

```
{
        "name": "Luke Skywalker",
        "height": "172",
        "mass": "77",
        "hair_color": "blond",
        "skin_color": "fair",
        "eye_color": "blue",
        "birth_year": "19BBY",
        "gender": "male",
        "homeworld": "https://swapi.dev/api/planets/1/",
        "films": [
                "https://swapi.dev/api/films/1/",
                "https://swapi.dev/api/films/2/",
                "https://swapi.dev/api/films/3/",
                "https://swapi.dev/api/films/6/"
        ],
        "species": [],
        "vehicles": [
                "https://swapi.dev/api/vehicles/14/",
                "https://swapi.dev/api/vehicles/30/"
        ],
        "starships": [
                "https://swapi.dev/api/starships/12/",
                "https://swapi.dev/api/starships/22/"
        ],
        "created": "2014-12-09T13:50:51.644000Z",
        "edited": "2014-12-20T21:17:56.891000Z",
        "url": "https://swapi.dev/api/people/1/"
}
```

To store the data in JSON, you can create a struct like this:

```
type Person struct {
        Name      string        `json:"name"`
        Height    string        `json:"height"`
        Mass      string        `json:"mass"`
        HairColor string        `json:"hair_color"`
        SkinColor string        `json:"skin_color"`
        EyeColor  string        `json:"eye_color"`
        BirthYear string        `json:"birth_year"`
```

```
        Gender    string        `json:"gender"`
        Homeworld string        `json:"homeworld"`
        Films     []string      `json:"films"`
        Species   []string      `json:"species"`
        Vehicles  []string      `json:"vehicles"`
        Starships []string      `json:"starships"`
        Created   time.Time     `json:"created"`
        Edited    time.Time     `json:"edited"`
        URL       string        `json:"url"`
}
```

The string literal after the definition in each field in the struct is called a *struct tag*. Go determines the mapping between the struct fields and the JSON elements using these struct tags. You don't need them if your mapping is exactly the same. However, as you can see, JSON normally uses snake case (variables with spaces replaced by underscores), with lowercase characters, while Go uses camel case (variables have no space but separation is indicated by a single capitalized letter).

As you can see from the struct, you can define string slices to store the arrays in the JSON and use something like `time.Time` as the data type as well. You can use most Go data types, and even maps (though only maps with strings as keys are supported).

Unmarshalling the data into the struct instance `person` is a single function call, using `json.Unmarshal`:

```
func unmarshal() (person Person) {
        file, err := os.Open("skywalker.json")
        if err != nil {
                log.Println("Error opening json file:", err)
        }
        defer file.Close()

        data, err := io.ReadAll(file)
        if err != nil {
                log.Println("Error reading json data:", err)
        }

        err = json.Unmarshal(data, &person)
        if err != nil {
                log.Println("Error unmarshalling json data:", err)
        }
        return
}
```

In this code, after reading the data from the file, you create a `Person` struct instance and then unmarshal the data into it using `json.Unmarshal`.

The JSON data came from the Star Wars API, so let's have a bit of fun and grab it directly from the API. You use the `http.Get` function and pass the URL in, but everything else is the same:

```
func unmarshalAPI() (person Person) {
        r, err := http.Get("https://swapi.dev/api/people/1")
        if err != nil {
                log.Println("Cannot get from URL", err)
        }
        defer r.Body.Close()

        data, err := io.ReadAll(r.Body)
        if err != nil {
                log.Println("Error reading json data:", err)
        }

        err = json.Unmarshal(data, &person)
        if err != nil {
                log.Println("Error unmarshalling json data:", err)
        }
        return
}
```

If you print out the `Person` struct instance, this is what you should get (the output is prettified):

```
json.Person{
    Name:      "Luke Skywalker",
    Height:    "172",
    Mass:      "77",
    HairColor: "blond",
    SkinColor: "fair",
    EyeColor:  "blue",
    BirthYear: "19BBY",
    Gender:    "male",
    Homeworld: "https://swapi.dev/api/planets/1/",
    Films:   {"https://swapi.dev/api/films/1/", "https://swapi.dev/api/films/2/",
    "https://swapi.dev/api/films/3/", "https://swapi.dev/api/films/6/"},
    Species:   {},
    Vehicles:  {"https://swapi.dev/api/vehicles/14/",
      "https://swapi.dev/api/vehicles/30/"},
    Starships: {"https://swapi.dev/api/starships/12/",
      "https://swapi.dev/api/starships/22/"},
    Created:   time.Date(2014, time.December, 9, 13, 50, 51, 644000000,
    time.UTC),
    Edited:    time.Date(2014, time.December, 20, 21, 17, 56, 891000000,
    time.UTC),
    URL:       "https://swapi.dev/api/people/1/",
}
```

9.2 Parsing Unstructured JSON Data

Problem

You want to parse some JSON data but you don't know the JSON data's structure or properties in advance enough to build structs, or the keys to the values are dynamic.

Solution

Use the same method as before but instead of predefined structs, use a map of strings to `any` to store the data.

Discussion

The structure of the Star Wars API is quite clear. However, this isn't always the case. Sometimes you just don't know the structure well enough to create structs, and documentation is not available. Also, sometimes keys to the values can be dynamic. Take a look at this JSON:

```
{
    "Luke Skywalker": [
        "https://swapi.dev/api/films/1/",
        "https://swapi.dev/api/films/2/",
        "https://swapi.dev/api/films/3/",
        "https://swapi.dev/api/films/6/"
      ],
    "C-3P0": [
        "https://swapi.dev/api/films/1/",
        "https://swapi.dev/api/films/2/",
        "https://swapi.dev/api/films/3/",
        "https://swapi.dev/api/films/4/",
        "https://swapi.dev/api/films/5/",
        "https://swapi.dev/api/films/6/"
      ],
    "R2D2": [
        "https://swapi.dev/api/films/1/",
        "https://swapi.dev/api/films/2/",
        "https://swapi.dev/api/films/3/",
        "https://swapi.dev/api/films/4/",
        "https://swapi.dev/api/films/5/",
        "https://swapi.dev/api/films/6/"
      ],
    "Darth Vader": [
        "https://swapi.dev/api/films/1/",
        "https://swapi.dev/api/films/2/",
        "https://swapi.dev/api/films/3/",
        "https://swapi.dev/api/films/6/"
      ]
}
```

Obviously from the JSON, the keys are not consistent and can change with the addition of a character. For such cases, how do you unmarshal the JSON data? Instead of predefined structs, you can use a map of strings to `any` (prior to Go 1.18 this would have been an empty interface—`interface{}`). Here's the code:

```
func unstructured() (output map[string]any) {
        file, err := os.Open("unstructured.json")
        if err != nil {
                log.Println("Error opening json file:", err)
        }
        defer file.Close()

        data, err := io.ReadAll(file)
        if err != nil {
                log.Println("Error reading json data:", err)
        }

        err = json.Unmarshal(data, &output)
        if err != nil {
                log.Println("Error unmarshalling json data:", err)
        }
        return
}
```

And here's the output:

```
map[string]any{
    "C-3P0": []any{
        "https://swapi.dev/api/films/1/",
        "https://swapi.dev/api/films/2/",
        "https://swapi.dev/api/films/3/",
        "https://swapi.dev/api/films/4/",
        "https://swapi.dev/api/films/5/",
        "https://swapi.dev/api/films/6/",
    },
    "Darth Vader": []any{
        "https://swapi.dev/api/films/1/",
        "https://swapi.dev/api/films/2/",
        "https://swapi.dev/api/films/3/",
        "https://swapi.dev/api/films/6/",
    },
    "Luke Skywalker": []any{
        "https://swapi.dev/api/films/1/",
        "https://swapi.dev/api/films/2/",
        "https://swapi.dev/api/films/3/",
        "https://swapi.dev/api/films/6/",
    },
    "R2D2": []any{
        "https://swapi.dev/api/films/1/",
        "https://swapi.dev/api/films/2/",
        "https://swapi.dev/api/films/3/",
        "https://swapi.dev/api/films/4/",
```

```
            "https://swapi.dev/api/films/5/",
            "https://swapi.dev/api/films/6/",
        },
    }
```

Try the same code on the earlier Luke Skywalker JSON data and see the output:

```
map[string]any{
    "birth_year": "19BBY",
    "created":    "2014-12-09T13:50:51.644000Z",
    "edited":     "2014-12-20T21:17:56.891000Z",
    "eye_color":  "blue",
    "films":      []any{
        "https://swapi.dev/api/films/1/",
        "https://swapi.dev/api/films/2/",
        "https://swapi.dev/api/films/3/",
        "https://swapi.dev/api/films/6/",
    },
    "gender":     "male",
    "hair_color": "blond",
    "height":     "172",
    "homeworld":  "https://swapi.dev/api/planets/1/",
    "mass":       "77",
    "name":       "Luke Skywalker",
    "skin_color": "fair",
    "species":    []any{
    },
    "starships": []any{
        "https://swapi.dev/api/starships/12/",
        "https://swapi.dev/api/starships/22/",
    },
    "url":      "https://swapi.dev/api/people/1/",
    "vehicles": []any{
        "https://swapi.dev/api/vehicles/14/",
        "https://swapi.dev/api/vehicles/30/",
    },
}
```

You might be thinking that this is much easier and simpler than trying to figure out the structs! Also, it's a lot more forgiving and flexible, so why not use this all the time? Using structs has its advantages. Using `any` essentially makes the data structure untyped. Structs can catch errors in the JSON where `any` simply lets them go.

It's a lot easier to retrieve data from structs than from a map because you know for sure what fields are available. Also, you need to do *type assertion* to get the data out of an interface. For example, you want to get the films that featured Darth Vader, so you might think you can do this:

```
unstruct := unstructured()
vader := unstruct["Darth Vader"]
first := vader[0]
```

You can't—you'll see this error instead:

```
invalid operation: vader[0] (type any does not support indexing)
```

This is because the variable `vader` is an `any`, so you have to type assert it first before you do anything:

```
unstruct := unstructured()
vader, ok := unstruct["Darth Vader"].([]any)
if !ok {
    log.Println("Cannot type assert")
}
first := vader[0]
```

You should normally try to use structs and map to `any` only as a last resort.

9.3 Parsing JSON Data Streams Into Structs

Problem

You want to parse JSON data from a stream.

Solution

Create structs to contain the JSON data. Create a decoder using `NewDecoder` in the `encoding/json` package, then call `Decode` on the decoder to decode data into the structs.

Discussion

Using `Unmarshal` is simple and straightforward for JSON files or API data. But what happens if the API is streaming JSON data? In that case, you can no longer use `Unmarshal` because `Unmarshal` needs to read the whole file at once. Instead, the `encoding/json` package provides a `Decoder` function for you to handle the data.

It might be difficult to understand the difference between JSON data and streaming JSON data, so take a look by comparing two different JSON files.

In this first JSON file you have an array of three JSON objects (part of the data is truncated to make it easier to read):

```
[{
"name": "Luke Skywalker",
"height": "172",
"mass": "77",
"hair_color": "blond",
"skin_color": "fair",
"eye_color": "blue",
"birth_year": "19BBY",
```

```
"gender": "male"
},
{
"name": "C-3PO",
"height": "167",
"mass": "75",
"hair_color": "n/a",
"skin_color": "gold",
"eye_color": "yellow",
"birth_year": "112BBY",
"gender": "n/a"
},
{
"name": "R2-D2",
"height": "96",
"mass": "32",
"hair_color": "n/a",
"skin_color": "white, blue",
"eye_color": "red",
"birth_year": "33BBY",
"gender": "n/a"
}]
```

To read this, you can use `Unmarshal` by unmarshalling into an array of `Person` structs:

```
func unmarshalStructArray() (people []Person) {
        file, err := os.Open("people.json")
        if err != nil {
                log.Println("Error opening json file:", err)
        }
        defer file.Close()

        data, err := io.ReadAll(file)
        if err != nil {
                log.Println("Error reading json data:", err)
        }

        err = json.Unmarshal(data, &people)
        if err != nil {
                log.Println("Error unmarshalling json data:", err)
        }
        return
}
```

This will result in an output like this:

```
[]json.Person{
    {
        Name:      "Luke Skywalker",
        Height:    "172",
        Mass:      "77",
        HairColor: "blond",
        SkinColor: "fair",
```

```
        EyeColor:  "blue",
        BirthYear: "19BBY",
        Gender:    "male",
        Homeworld: "",
        Films:     nil,
        Species:   nil,
        Vehicles:  nil,
        Starships: nil,
        Created:   time.Date(1, time.January, 1, 0, 0, 0, 0, time.UTC),
        Edited:    time.Date(1, time.January, 1, 0, 0, 0, 0, time.UTC),
        URL:       "",
    },
    {
        Name:      "C-3PO",
        Height:    "167",
        Mass:      "75",
        HairColor: "n/a",
        SkinColor: "gold",
        EyeColor:  "yellow",
        BirthYear: "112BBY",
        Gender:    "n/a",
        Homeworld: "",
        Films:     nil,
        Species:   nil,
        Vehicles:  nil,
        Starships: nil,
        Created:   time.Date(1, time.January, 1, 0, 0, 0, 0, time.UTC),
        Edited:    time.Date(1, time.January, 1, 0, 0, 0, 0, time.UTC),
        URL:       "",
    },
    {
        Name:      "R2-D2",
        Height:    "96",
        Mass:      "32",
        HairColor: "n/a",
        SkinColor: "white, blue",
        EyeColor:  "red",
        BirthYear: "33BBY",
        Gender:    "n/a",
        Homeworld: "",
        Films:     nil,
        Species:   nil,
        Vehicles:  nil,
        Starships: nil,
        Created:   time.Date(1, time.January, 1, 0, 0, 0, 0, time.UTC),
        Edited:    time.Date(1, time.January, 1, 0, 0, 0, 0, time.UTC),
        URL:       "",
    },
}
```

This is an array of `Person` structs, which you get after unmarshalling a single JSON array. However, when you get a stream of JSON objects, this is no longer possible. Here is another JSON file, one that is representative of a JSON data stream:

```
{
"name": "Luke Skywalker",
"height": "172",
"mass": "77",
"hair_color": "blond",
"skin_color": "fair",
"eye_color": "blue",
"birth_year": "19BBY",
"gender": "male"
}
{
"name": "C-3PO",
"height": "167",
"mass": "75",
"hair_color": "n/a",
"skin_color": "gold",
"eye_color": "yellow",
"birth_year": "112BBY",
"gender": "n/a"
}
{
"name": "R2-D2",
"height": "96",
"mass": "32",
"hair_color": "n/a",
"skin_color": "white, blue",
"eye_color": "red",
"birth_year": "33BBY",
"gender": "n/a"
}
```

Notice that this is not a single JSON object but three consecutive JSON objects. This is no longer a valid JSON file, but it's something you can get when you read the `Body` of a `http.Response` struct. If you try to read this using `Unmarshal` you will get an error:

```
Error unmarshalling json data: invalid character '{' after top-level value
```

However, you can parse it by decoding it using a `Decoder`:

```
func decode(p chan Person) {
        file, err := os.Open("people_stream.json")
        if err != nil {
                log.Println("Error opening json file:", err)
        }
        defer file.Close()

        decoder := json.NewDecoder(file)
```

```
        for {
                var person Person
                err = decoder.Decode(&person)
                if err == io.EOF {
                        break
                }
                if err != nil {
                        log.Println("Error decoding json data:", err)
                        break
                }
                p <- person
        }
        close(p)
}
```

First, you create a decoder using `json.NewDecoder` and passing it the reader, in this case, it's the file you read from. Then while you're looping in the `for` loop, you call `Decode` on the decoder, passing it the struct you want to store the data in. If all goes well, every time it loops, a new `Person` struct instance is created from the data. You can use the data then. If there is no more data in the reader, i.e., you hit `io.EOF`, you'll break from the `for` loop.

In the case of the preceding code, you pass in a channel, in which you store the `Person` struct instance in every loop. When you're done reading all the JSON in the file, you'll close the channel:

```
func main() {
        p := make(chan Person)
        go decode(p)
        for {
                person, ok := <-p
                if ok {
            fmt.Printf("%# v\n", pretty.Formatter(person))
                } else {
                        break
                }
        }
}
```

Here's the output from the code:

```
json.Person{
    Name:      "Luke Skywalker",
    Height:    "172",
    Mass:      "77",
    HairColor: "blond",
    SkinColor: "fair",
    EyeColor:  "blue",
    BirthYear: "19BBY",
    Gender:    "male",
    Homeworld: "",
    Films:     nil,
```

```
        Species:   nil,
        Vehicles:  nil,
        Starships: nil,
        Created:   time.Date(1, time.January, 1, 0, 0, 0, 0, time.UTC),
        Edited:    time.Date(1, time.January, 1, 0, 0, 0, 0, time.UTC),
        URL:       "",
    }
    json.Person{
        Name:      "C-3PO",
        Height:    "167",
        Mass:      "75",
        HairColor: "n/a",
        SkinColor: "gold",
        EyeColor:  "yellow",
        BirthYear: "112BBY",
        Gender:    "n/a",
        Homeworld: "",
        Films:     nil,
        Species:   nil,
        Vehicles:  nil,
        Starships: nil,
        Created:   time.Date(1, time.January, 1, 0, 0, 0, 0, time.UTC),
        Edited:    time.Date(1, time.January, 1, 0, 0, 0, 0, time.UTC),
        URL:       "",
    }
    json.Person{
        Name:      "R2-D2",
        Height:    "96",
        Mass:      "32",
        HairColor: "n/a",
        SkinColor: "white, blue",
        EyeColor:  "red",
        BirthYear: "33BBY",
        Gender:    "n/a",
        Homeworld: "",
        Films:     nil,
        Species:   nil,
        Vehicles:  nil,
        Starships: nil,
        Created:   time.Date(1, time.January, 1, 0, 0, 0, 0, time.UTC),
        Edited:    time.Date(1, time.January, 1, 0, 0, 0, 0, time.UTC),
        URL:       "",
    }
```

You can see that three `Person` structs are being printed here, one after another, as opposed to the earlier one that was an array of `Person` structs.

A question that sometimes arises is when should you use `Unmarshal` and when should you use `Decode`?

`Unmarshal` is easier to use for a single JSON object, but it won't work when you have a stream of them coming in from a reader. Also, its simplicity means it's not as flexible; you just get the whole JSON data at a go.

`Decode`, on the other hand, works well for both single JSON objects and streaming JSON data. Also, with `Decode` you can do stuff with the JSON at a finer level without needing to get the entire JSON data out first. This is because you can inspect the JSON as it comes in, even at a token level. The only slight drawback is that it is more verbose.

In addition, `Decode` is a bit faster. You can do a quick benchmarking test on both:

```
var luke []byte = []byte(
        `{
        "name": "Luke Skywalker",
        "height": "172",
        "mass": "77",
        "hair_color": "blond",
        "skin_color": "fair",
        "eye_color": "blue",
        "birth_year": "19BBY",
        "gender": "male"
}`)

func BenchmarkUnmarshal(b *testing.B) {
        var person Person
        b.ResetTimer()
        for i := 0; i < b.N; i++ {
                json.Unmarshal(luke, &person)
        }
}

func BenchmarkDecode(b *testing.B) {
        var person Person
        data := bytes.NewReader(luke)
        decoder := json.NewDecoder(data)
        b.ResetTimer()
        for i := 0; i < b.N; i++ {
                decoder.Decode(&person)
                data.Seek(0, 0)
        }
}
```

Here you're benchmarking `Unmarshal` and `Decode` using the standard Go benchmarking tool. To make sure you're benchmarking properly you reset the timer just before you run the iterations that test the performance of `Unmarshal` and `Decode`. You place the creation of the decoder before the benchmarking because you need to create the decoder only once since it's going to wrap around the reader that's streaming data in. However, once `Decode` is called, you need to move the offset to the start for the next benchmarking loop.

Run this in the command line to start the benchmarking:

```
$ go test -bench=. -benchmem
```

And this is the result:

```
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch10_json
BenchmarkUnmarshal-8      437274          2494 ns/op     272 B/op   12 allocs/op
BenchmarkDecode-8         486051          2368 ns/op      48 B/op    8 allocs/op
PASS
ok      github.com/sausheong/gocookbook/ch10_json       6.242s
```

As you can see, `Decode` is only a bit faster, taking 2,258 ns/op (nanoseconds per operation) while `Unmarshal` takes 2,418 ns/op. However, `Decode` uses only 48 B/op (bytes per operation), considerably less than `Unmarshal`, which uses 272 B/op.

9.4 Creating JSON Data Byte Arrays from Structs

Problem

You want to create JSON data from a struct.

Solution

Create the structs then use the `json.Marshal` package to marshal the data into a JSON slice of bytes.

Discussion

Creating JSON data is essentially the reverse of parsing it:

1. Create structs that you will use to marshal data from.
2. Marshal the data into a JSON string using `json.Marshal` or `json.Marshal Indent`.

You will be reusing the same structs as the previous recipe for parsing JSON. You'll also use the function used to unmarshal JSON data from the Star Wars API:

```
func main() {
        person := get("https://swapi.dev/api/people/14")

        data, err := json.Marshal(&person)
        if err != nil {
                log.Println("Cannot marshal person:", err)
        }
        err = os.WriteFile("han.json", data, 0644)
        if err != nil {
```

```
                log.Println("Cannot write to file", err)
        }
}

func get(url string) Person {
        r, err := http.Get(url)
        if err != nil {
                log.Println("Cannot get from URL", err)
        }
        defer r.Body.Close()

        data, err := os.ReadAll(r.Body)
        if err != nil {
                log.Println("Error reading json data:", err)
        }

        var person Person
        json.Unmarshal(data, &person)
        return person
}
```

The `get` function returns a `Person` struct instance that you can use for marshaling to a file. The `json.Marshal` function takes the data in the struct instance and returns a byte slice, `data`, containing the JSON string. If you just want it as a string, you can cast it to a string and use it. Here, you pass it on to `os.WriteFile` to create a new JSON file:

```
{"name":"Han Solo","height":"180","mass":"80","hair_color":"brown",
"skin_color":"fair","eye_color":"brown","birth_year":"29BBY","gender":"male",
"homeworld":"https://swapi.dev/api/planets/22/","films":
["https://swapi.dev/api/films/1/","https://swapi.dev/api/films/2/",
"https://swapi.dev/api/films/3/"],"species":[],"vehicles":[],"starships":
["https://swapi.dev/api/starships/10/","https://swapi.dev/api/starships/22/"],
"created":"2014-12-10T16:49:14.582Z","edited":"2014-12-20T21:17:50.334Z",
"url":"https://swapi.dev/api/people/14/"}
```

This is not very readable. If you want a more readable version, you can use `json.MarshalIndent` instead. You need to put in two more parameters: the first is the prefix, and the second is the indent. Mostly if you want to have a clean JSON output, the prefix is an empty string while the indent is a single space:

```
data, err := json.MarshalIndent(&person, "", " ")
```

This will produce a more readable version:

```
{
 "name": "Han Solo",
 "height": "180",
 "mass": "80",
 "hair_color": "brown",
 "skin_color": "fair",
 "eye_color": "brown",
 "birth_year": "29BBY",
 "gender": "male",
 "homeworld": "https://swapi.dev/api/planets/22/",
 "films": [
  "https://swapi.dev/api/films/1/",
  "https://swapi.dev/api/films/2/",
  "https://swapi.dev/api/films/3/"
 ],
 "species": [],
 "vehicles": [],
 "starships": [
  "https://swapi.dev/api/starships/10/",
  "https://swapi.dev/api/starships/22/"
 ],
 "created": "2014-12-10T16:49:14.582Z",
 "edited": "2014-12-20T21:17:50.334Z",
 "url": "https://swapi.dev/api/people/14/"
}
```

9.5 Creating JSON Data Streams from Structs

Problem

You want to create streaming JSON data from structs.

Solution

Create an encoder using `NewEncoder` in the `encoding/json` package, passing it an `io.Writer`. Then call `Encode` on the encoder to encode structs data to a stream.

Discussion

The `io.Writer` interface has a `Write` method that writes bytes to the underlying data stream. You use `NewEncoder` to create an encoder that wraps around a writer. When you call `Encode` on the encoder, it will write the JSON struct instance onto the writer.

To show this properly you'll need some JSON structs. You'll use the same Star Wars people API as before to create the structs:

```
func get(n int) (person Person) {
        r, err := http.Get("https://swapi.dev/api/people/" + strconv.Itoa(n))
        if err != nil {
                log.Println("Cannot get from URL", err)
        }
        defer r.Body.Close()

        data, err := ioutil.ReadAll(r.Body)
        if err != nil {
                log.Println("Error reading json data:", err)
        }

        json.Unmarshal(data, &person)
        return
}
```

This `get` function will call the API and return the requested `Person` struct. You'll need to use this `Person` struct instance next:

```
func main() {
        encoder := json.NewEncoder(os.Stdout)
        for i := 1; i < 4; i++ { // we're just retrieving 3 records
                person := get(i)
                encoder.Encode(person)
        }
}
```

As you can see, you're using `os.Stdout` as the writer. Actually `os.Stdout` is an `os.File` struct instance but a `File` is also a writer, so that's fine. What this does is write the encoding to `os.Stdout` one at a time.

First, you create an encoder using `json.NewEncoder`, passing it `os.Stdout` as the writer. Next, as you loop you get a `Person` struct instance and pass that to `Encode` to write to `os.Stdout`.

When you run the program, you should see something like this, but each JSON encoding will appear one by one:

```
{"name":"Luke Skywalker","height":"172","mass":"77","hair_color":"blond",
"skin_color":"fair","eye_color":"blue","birth_year":"19BBY","gender":"male",
"homeworld":"https://swapi.dev/api/planets/1/","films":
["https://swapi.dev/api/films/1/","https://swapi.dev/api/films/2/",
"https://swapi.dev/api/films/3/","https://swapi.dev/api/films/6/"],
"species":[],"vehicles":["https://swapi.dev/api/vehicles/14/",
"https://swapi.dev/api/vehicles/30/"],"starships":
["https://swapi.dev/api/starships/12/","https://swapi.dev/api/starships/22/"],
"created":"2014-12-09T13:50:51.644Z","edited":"2014-12-20T21:17:56.891Z",
"url":"https://swapi.dev/api/people/1/"}
{"name":"C-3PO","height":"167","mass":"75","hair_color":"n/a","skin_color":
"gold","eye_color":"yellow","birth_year":"112BBY","gender":"n/a","homeworld":
"https://swapi.dev/api/planets/1/","films":["https://swapi.dev/api/films/1/",
"https://swapi.dev/api/films/2/","https://swapi.dev/api/films/3/",
```

```
"https://swapi.dev/api/films/4/","https://swapi.dev/api/films/5/",
"https://swapi.dev/api/films/6/"],"species":["https://swapi.dev/api/species/2/"],
"vehicles":[],"starships":[],"created":"2014-12-10T15:10:51.357Z","edited":
"2014-12-20T21:17:50.309Z","url":"https://swapi.dev/api/people/2/"}
{"name":"R2-D2","height":"96","mass":"32","hair_color":"n/a","skin_color":
"white, blue","eye_color":"red","birth_year":"33BBY","gender":"n/a",
"homeworld":"https://swapi.dev/api/planets/8/","films":
["https://swapi.dev/api/films/1/","https://swapi.dev/api/films/2/",
"https://swapi.dev/api/films/3/","https://swapi.dev/api/films/4/",
"https://swapi.dev/api/films/5/","https://swapi.dev/api/films/6/"],
"species":["https://swapi.dev/api/species/2/"],"vehicles":[],"starships":[],
"created":"2014-12-10T15:11:50.376Z","edited":"2014-12-20T21:17:50.311Z",
"url":"https://swapi.dev/api/people/3/"}
```

If you are annoyed by the untidy output here and wonder if there is an equivalent of `MarshalIndent`, yes there is. Just set up the encoder with `SetIndent` like this and you're good to go:

```
encoder.SetIndent("", " ")
```

You might be wondering what the difference is between using `Encode` and using `Marshal`. To use `Marshal` you need to put everything into an object and marshal it all at once—you can't stream the JSON encodings one at a time.

In other words, if you have JSON structs coming to you and you either don't know when it will all come or if you want to write the JSON encodings out first, then you need to use `Encode`. You can use `Marshal` only if you have all the JSON data available to you.

And of course, `Encode` is also faster than `Marshal`, as this benchmarking shows:

```
var jsonBytes []byte = []byte(jsonString)
var person Person

func BenchmarkMarshal(b *testing.B) {
    json.Unmarshal(jsonBytes, &person)
    b.ResetTimer()
        for i := 0; i < b.N; i++ {
                data, _ := json.Marshal(person)
                io.Discard.Write(data)
        }
}

func BenchmarkEncoder(b *testing.B) {
        json.Unmarshal(jsonBytes, &person)
        b.ResetTimer()
    encoder := json.NewEncoder(io.Discard)
        for i := 0; i < b.N; i++ {
                encoder.Encode(person)
        }
}
```

You need to prepare the JSON struct instance before the test, so you unmarshal the data into a `Person` struct instance as part of setting up, before running the benchmark loop. Also use `io.Discard` as the writer instead. The `io.Discard` is a writer on which all write calls will succeed and is the most convenient to use here.

To benchmark `Marshal` you marshal the `Person` struct instance and then write it to `io.Discard`. To benchmark `Encode`, create an encoder that wraps around `io.Discard` and then encode the `Person` struct instance to it. As in the decoder benchmarking, you placed the creation of the encoder before the iterations because you need to create it only once.

This is the result of the benchmark:

```
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch10_json
BenchmarkMarshal-10      4717722     236.2 ns/op     504 B/op    3 allocs/op
BenchmarkEncoder-10      5885935     203.4 ns/op     264 B/op    2 allocs/op
PASS
ok      github.com/sausheong/gocookbook/ch10_json       4.345s
```

As before, `Encode` is faster, and it also uses less memory, taking up less than half—about 128 B/op as compared to `Marshal`, which takes up 288 B/op.

9.6 Omitting Fields in Structs

Problem

When marshaling JSON structs as JSON encoding, sometimes there is no data for some of the struct variables. You want to create JSON encoding that leaves out the variables without any data.

Solution

Use the `omitempty` tag to define struct variables that can be omitted when marshaling.

Discussion

Take a look at the `Person` struct again:

```
type Person struct {
        Name      string        `json:"name"`
        Height    string        `json:"height"`
        Mass      string        `json:"mass"`
        HairColor string        `json:"hair_color"`
        SkinColor string        `json:"skin_color"`
        EyeColor  string        `json:"eye_color"`
```

```
        BirthYear string        `json:"birth_year"`
        Gender    string        `json:"gender"`
        Homeworld string        `json:"homeworld"`
        Films     []string      `json:"films"`
        Species   []string      `json:"species"`
        Vehicles  []string      `json:"vehicles"`
        Starships []string      `json:"starships"`
        Created   time.Time     `json:"created"`
        Edited    time.Time     `json:"edited"`
        URL       string        `json:"url"`
}
```

You might notice that the API doesn't specify the species when the person is a human and that many of the characters don't have vehicles or starships tagged to them. So when you marshal the struct, it will come out as an empty array:

```
{
 "name": "Owen Lars",
 "height": "178",
 "mass": "120",
 "hair_color": "brown, grey",
 "skin_color": "light",
 "eye_color": "blue",
 "birth_year": "52BBY",
 "gender": "male",
 "homeworld": "https://swapi.dev/api/planets/1/",
 "films": [
  "https://swapi.dev/api/films/1/",
  "https://swapi.dev/api/films/5/",
  "https://swapi.dev/api/films/6/"
 ],
 "species": [],
 "vehicles": [],
 "starships": [],
 "created": "2014-12-10T15:52:14.024Z",
 "edited": "2014-12-20T21:17:50.317Z",
 "url": "https://swapi.dev/api/people/6/"
}
```

If you don't want to show the species, vehicles, or starships, you can use the `omit empty` tag on the JSON struct tags:

```
Species   []string      `json:"species,omitempty"`
Vehicles  []string      `json:"vehicles,omitempty"`
Starships []string      `json:"starships,omitempty"`
```

When you run the same code again, you will no longer see them in the output:

```
{
 "name": "Owen Lars",
 "height": "178",
 "mass": "120",
 "hair_color": "brown, grey",
 "skin_color": "light",
 "eye_color": "blue",
 "birth_year": "52BBY",
 "gender": "male",
 "homeworld": "https://swapi.dev/api/planets/1/",
 "films": [
  "https://swapi.dev/api/films/1/",
  "https://swapi.dev/api/films/5/",
  "https://swapi.dev/api/films/6/"
 ],
 "created": "2014-12-10T15:52:14.024Z",
 "edited": "2014-12-20T21:17:50.317Z",
 "url": "https://swapi.dev/api/people/6/"
}
```

Why would you want to do this at all? In this API, height and mass are strings. However, if they are integers, and you don't know their height or mass, the default value would be 0. In this case, these values are wrong, but sometimes they might be correct as well (say, for example, a Jedi Force ghost would have neither height nor mass), but you can't tell which is which. In this case, not showing it at all is the better option.

CHAPTER 10

Binary Recipes

10.0 Introduction

So far our I/O-related recipes have been working with text data like in CSV or JSON. This is good because this data is meant to be read by humans as well as machines. However, verbose text data formats can sometimes be a disadvantage.

Speed, of course, is one consideration—it's faster to transfer less data and being less verbose helps. With the proliferation of Internet of Things (IoT) devices and sensors, we often need to resort to low-bandwidth networks to send data. Memory and storage space is the other consideration. Smaller devices and sensors, often powered by batteries, mean that you cannot afford to use a lot of memory or storage for the data.

All this comes down to compacting data formats to the bit and byte levels. There are many such formats in existence already—storing data in smaller sizes has always been a necessity for past computing. Some recently popular formats include BSON (Binary JSON), Protocol Buffers/protobuf from Google, and Apache Thrift from Facebook.

In this chapter, you'll go through Go's binary format, gob. You'll also build your custom binary format using the `encoding/binary` package. In both cases, this chapter discusses how you can encode, store, and decode these formats.

10.1 Encoding Data to gob Format Data

Problem

You want to encode structs into binary gob format.

Solution

Use the `encoding/gob` package to encode the structs into bytes that can be stored or sent elsewhere.

Discussion

The `encoding/gob` package is a Go library to encode and decode a binary format. The data can be anything but is particularly useful with Go structs. You should be aware that gob is a proprietary Go binary format. Although there are attempts to decode gob in other languages, it is not a widely used format like protobuf or Thrift. It is advisable to use a more commonly used format if you have more complex use cases for binary data.

Take a look at an example. You want to deploy small electricity meters all over a building to measure the consumption of energy in the building. The metering points are not only per floor but also per unit, per area, and even per meeting room. They will also be used in common areas, including lighting as well as larger loads like lifts and escalators. The information you gather will help you monitor any abnormal usage of electricity, identify wastage, and allocate costs to the different occupants of the building. You are going to deploy a lot of these meters all over the building so the communications use a low-powered wide area network (LP-WAN). Your requirement is for data packets to be small so they can be transported through the LP-WAN efficiently.

Start with a simple struct to capture the information from the meter:

```
type Meter struct {
        Id        uint32
        Voltage   uint8
        Current   uint8
        Energy    uint32
        Timestamp uint64
}
```

You set a unique identifier for each meter; the voltage, current, and energy are what is being measured; and the timestamp gives you the time the reading is taken. The voltage and current are measured at the moment, but the energy in kilowatt-hours is how much energy has been consumed since the meter started.

Next, see how you can take readings from the meter and write them to a stream to be sent across the LP-WAN. For this, you will assume the following meter-reading data will be available and construct a struct to contain the data:

```
var reading Meter = Meter{
        Id:        123456,
        Voltage:   229.5,
        Current:   1.3,
        Energy:    4321,
        Timestamp: uint64(time.Now().UnixNano()),
}
```

You also use a file to represent the network and will write to it:

```
func write(data interface{}, filename string) {
        file, err := os.Create("reading")
        if err != nil {
                log.Println("Cannot create file:", err)
        }
        encoder := gob.NewEncoder(file)
        err = encoder.Encode(data)
        if err != nil {
                log.Println("Cannot encode data to file:", err)
        }
}
```

First, you create a file named *reading*, which will be your `Writer`. You then create an encoder around this writer and call `Encode` on it, passing it the struct instance. This will encode the struct instance in the gob format and will write it to a file.

10.2 Decoding gob Format Data to Structs

Problem

You want to decode gob format data back to structs.

Solution

Use the `encoding/gob` package to decode the gob format data back to structs.

Discussion

Recipe 10.1 showed how to create gob format binary data from a struct. Re-creating the struct instance from the gob data is very similar, except that you use `Decode` instead:

```
func read(data interface{}, filename string) {
        file, err := os.Open("reading")
        if err != nil {
                log.Println("Cannot read file:", err)
```

```
        }
        decoder := gob.NewDecoder(file)
        err = decoder.Decode(data)
        if err != nil {
                log.Println("Cannot decode data:", err)
        }
}
```

Open the file named *reading*, which you created from the previous recipe. This file will be your `Reader`. You will create a decoder around the reader and then call `Decode` on it, passing the struct instance to be populated with the data.

Call the `read` function and pass in a reference to a struct instance:

```
read(&reading, "reading")
```

The `reading` struct instance will be populated with the data after the call.

Now that you can write and read gob format, how does it compare with doing this with JSON? First, the data setup for the benchmark tests:

```
type Meter struct {
        Id        uint32
        Voltage   float32
        Current   float32
        Energy    uint32
        Timestamp uint64
}

var reading Meter = Meter{
        Id:        123456,
        Voltage:   229.5,
        Current:   1.3,
        Energy:    4321,
        Timestamp: uint64(time.Now().UnixNano()),
}

var jsonString string = `{
        "Id": 123456,
        "Voltage": 229.5,
        "Current": 1.3,
        "Energy": 4321,
        "Timestamp": "2009-11-10 23:00:00 +0000 UTC"
        }`

var jsonData []byte = []byte(jsonString)
var gobData *bytes.Buffer

func init() {
        gobData = bytes.NewBuffer([]byte{})
        gob.NewEncoder(gobData).Encode(reading)
}
```

You have a struct instance `reading` to be used for encoding for both JSON and gob, and `jsonData` and `gobData` slices to be used for decoding for JSON and gob, respectively. The `init` function prepares the `gobData` in advance for decoding.

Now look at the benchmarking tests:

```
func BenchmarkEncodeJson(b *testing.B) {
        encoder := json.NewEncoder(io.Discard)
        b.ResetTimer()
        for i := 0; i < b.N; i++ {
                encoder.Encode(reading)
        }
}

func BenchmarkEncodeGob(b *testing.B) {
        encoder := gob.NewEncoder(io.Discard)
        b.ResetTimer()
        for i := 0; i < b.N; i++ {
                encoder.Encode(reading)
        }
}

func BenchmarkDecodeJson(b *testing.B) {
        var data Meter
        buf := bytes.NewReader(jsonData)
        decoder := json.NewDecoder(buf)
        b.ResetTimer()
        for i := 0; i < b.N; i++ {
                decoder.Decode(&data)
                buf.Seek(0, 0)
        }
}

func BenchmarkDecodeGob(b *testing.B) {
        var data Meter
        buf := bytes.NewReader(gobData.Bytes())
        decoder := gob.NewDecoder(buf)
        b.ResetTimer()
        for i := 0; i < b.N; i++ {
                decoder.Decode(&data)
                buf.Seek(0, 0)
        }
}
```

There's nothing fancy here; each benchmark is as simple as possible to test the encoding and decoding of JSON and gob formats. Here are the results:

```
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch10_binary/benchmarks
BenchmarkEncodeJson-10      4835588    241.2 ns/op     24 B/op    1 allocs/op
BenchmarkEncodeGob-10       7738795    155.4 ns/op     24 B/op    1 allocs/op
BenchmarkDecodeJson-10      1000000     1023 ns/op    101 B/op    5 allocs/op
BenchmarkDecodeGob-10      13910449    81.27 ns/op     96 B/op    2 allocs/op
PASS
ok      github.com/sausheong/gocookbook/ch10_binary/benchmarks  5.138s
```

Encoding gob, as you can see, is faster than encoding JSON, though the amount of memory used is the same. Decoding gob is much faster than decoding JSON as well and uses a lot less memory.

10.3 Encoding Data to a Customized Binary Format

Problem

You want to encode struct data to a customized binary format.

Solution

Design your customized format and use the `encoding/binary` package to write data in structs to it.

Discussion

Using gob has a couple of drawbacks. First, gob is supported by Go only and works best if both sender and receiver are written in Go. Second, gob stores the whole struct, labels and all, which makes the encoded binary data relatively large. In fact, there is no difference between the size of a piece of JSON data compared to the size of a piece of gob data when they have the same content!

An alternative is to strip away the labels; for example, the `Meter` struct can be stored this way. Figure 10-1 shows how data for the `Meter` struct will be stored.

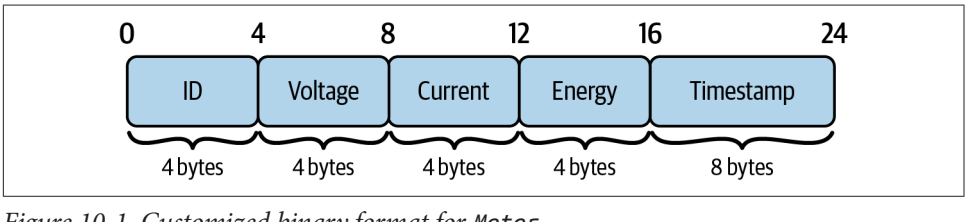

Figure 10-1. Customized binary format for `Meter`

Remember `uint8` is 1 byte, `uint16` is 2 bytes, `uint32` is 4 bytes, and `uint64` is 8 bytes. You don't need the labels if you know the positions of the values.

Writing this format is surprisingly easy. You simply use `binary.Write` to write the data from the struct instance into this format:

```
func main() {
        file, err := os.Create("data.bin")
        if err != nil {
                log.Println("Cannot create file:", err)
        }
        err = binary.Write(file, binary.BigEndian, reading)
        if err != nil {
                log.Println("Cannot write to file:", err)
        }
}
```

The first parameter is the writer you want to write to; in this case, it's a file. The second parameter is the byte order for format. The `encoding/binary` supports both big-endian and little-endian, and in this case, you are using big-endian. The last parameter is the struct instance you're taking the data from.

If you look at the file that's created, it's just 24 bytes, as opposed to the earlier gob format, which turned out to be 110 bytes. That's a significant reduction if you're moving smaller packets of data over a low-bandwidth network.

You can check out the performance next. The benchmarking test is quite simple:

```
func BenchmarkEncodeBinary(b *testing.B) {
        for i := 0; i < b.N; i++ {
                binary.Write(io.Discard, binary.BigEndian, reading)
        }
}
```

Run it against the other encoding benchmarks to see how fast they run:

```
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch10_binary/benchmarks
BenchmarkEncodeJson-10      4699872    238.6 ns/op    24 B/op    1 allocs/op
BenchmarkEncodeGob-10       7612948    156.8 ns/op    24 B/op    1 allocs/op
BenchmarkEncodeBinary-10    5537563    216.9 ns/op    88 B/op    7 allocs/op
PASS
ok      github.com/sausheong/gocookbook/ch10_binary/benchmarks  4.646s
```

You might have thought it would encode faster but it's only slightly better than encoding in JSON, and gob encoding beats it by quite a bit. In addition, it takes up more memory doing the job.

While `binary.Write` is the easiest way to encode customized binary data, you can also use the `encoding/binary` package to encode a struct instance manually. Here's how this can be done:

```
func main() {
        file, err := os.Create("data.bin")
        if err != nil {
                log.Println("Cannot create file:", err)
        }
        defer file.Close()

        buf := make([]byte, 24)
        binary.BigEndian.PutUint32(buf[0:], reading.Id)
        binary.BigEndian.PutUint32(buf[4:], math.Float32bits(reading.Voltage))
        binary.BigEndian.PutUint32(buf[8:], math.Float32bits(reading.Current))
        binary.BigEndian.PutUint32(buf[12:], reading.Energy)
        binary.BigEndian.PutUint64(buf[16:], reading.Timestamp)
        file.Write(buf)
}
```

First, you need to create a byte array. You know the format is 24 bytes so you make the byte array correctly sized. Then you use the correct function in the `binary.BigEndian` struct to put the values in the struct instance into the various positions in the byte array. For example, `reading.Id` is a `unit32` so you use the `PutUint32` method to place the values in the byte array at the correct location.

The values in `reading.Voltage` and `reading.Current` are `float32` so you need to convert the value into the correct IEEE 753 binary representation first, before placing it into the byte array.

Finally, when you are done, you write the byte array to the file.

It seems like a lot of work, and the file size remains the same, so check out the performance:

```
func BenchmarkEncodeBinaryManual(b *testing.B) {
        for i := 0; i < b.N; i++ {
                buf := make([]byte, 4+4+4+4+8)
                binary.BigEndian.PutUint32(buf[0:], reading.Id)
                binary.BigEndian.PutUint32(buf[4:], math.Float32bits
                (reading.Voltage))
                binary.BigEndian.PutUint32(buf[8:], math.Float32bits
                (reading.Current))
                binary.BigEndian.PutUint32(buf[12:], reading.Energy)
                binary.BigEndian.PutUint64(buf[16:], reading.Timestamp)
                io.Discard.Write(buf)
        }
}
```

Now run the benchmark:

```
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch10_binary/benchmarks
BenchmarkEncodeJson-10            4720197   238.2 ns/op   24 B/op   1 allocs/op
BenchmarkEncodeGob-10             7697342   156.9 ns/op   24 B/op   1 allocs/op
BenchmarkEncodeBinary-10          5449068   217.5 ns/op   88 B/op   7 allocs/op
BenchmarkEncodeBinaryManual-10   78357111   15.28 ns/op   24 B/op   1 allocs/op
PASS
ok      github.com/sausheong/gocookbook/ch10_binary/benchmarks  6.798s
```

It's a world of difference! The performance is significantly better, and it uses less memory than the `Write` alone.

10.4 Decoding Data with a Customized Binary Format to Structs

Problem

You want to decode the customized binary format back to structs.

Solution

Use the `encoding/binary` package to take data from the binary format and reconstruct structs from it.

Discussion

Decoding customized binary data into structs works the same way you'd expect:

```
func main() {
        var data Meter
        file, err := os.Open("bindata")
        if err != nil {
                log.Println("Cannot read file:", err)
        }
        err = binary.Read(file, binary.BigEndian, &data)
        if err != nil {
                log.Println("Cannot read binary:", err)
        }
}
```

Start with creating a struct to contain your data. Instead of `Write` you use `Read`, passing the reader to it, the byte order, and the struct instance you want to store data. This will read the bytes from the file and write them to the struct.

Here's how you can do it manually, instead of using a one-liner like `Read`:

```
func main() {
        var data Meter = Meter{}
        file, err := os.Open("data.bin")
        if err != nil {
                log.Println("Cannot read file:", err)
        }
        buf := make([]byte, 24)
        file.Read(buf)
        defer file.Close()

        data.Id = binary.BigEndian.Uint32(buf[:4])
        data.Voltage = math.Float32frombits(binary.BigEndian.Uint32(buf[4:8]))
        data.Current = math.Float32frombits(binary.BigEndian.Uint32(buf[8:12]))
        data.Energy = binary.BigEndian.Uint32(buf[12:16])
        data.Timestamp = binary.BigEndian.Uint64(buf[16:])
        fmt.Println(data)
}
```

First, you read the data from the file into a byte array. Then using the functions from `binary.BigEndian` you extract the data from the byte array and rebuild a new struct using the data.

It's pretty straightforward so you would expect the performance to be good as well. To check that out, compare the performance of reading with a single function, `Read`, to doing it manually:

```
func BenchmarkDecodeBinary(b *testing.B) {
        var data Meter
        buf := bytes.NewReader(binData.Bytes())
        b.ResetTimer()
        for i := 0; i < b.N; i++ {
                binary.Read(buf, binary.BigEndian, &data)
                buf.Seek(0, 0)
        }
}

func BenchmarkDecodeBinaryManual(b *testing.B) {
        var data Meter
        buf := binData.Bytes()
        b.ResetTimer()
        for i := 0; i < b.N; i++ {
                data.Id = binary.BigEndian.Uint32(buf[:4])
                data.Voltage = math.Float32frombits(binary.BigEndian.Uint32
                (buf[4:8]))
                data.Current = math.Float32frombits(binary.BigEndian.Uint32
                (buf[8:12]))
                data.Energy = binary.BigEndian.Uint32(buf[12:16])
                data.Timestamp = binary.BigEndian.Uint64(buf[16:])
        }
}
```

Remember for the single function call benchmark, you need to rewind the reader, but the manual benchmark doesn't need to rewind because you're just reading from the byte slice.

These are the benchmark results:

```
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch10_binary/benchmarks
BenchmarkDecodeJson-10            1141518       1032 ns/op 101 B/op  5 allocs/op
BenchmarkDecodeGob-10            14736854      81.21 ns/op  96 B/op  2 allocs/op
BenchmarkDecodeBinary-10         14016166      85.29 ns/op  24 B/op  1 allocs/op
BenchmarkDecodeBinaryManual-10 1000000000      1.099 ns/op   0 B/op  0 allocs/op
PASS
ok      github.com/sausheong/gocookbook/ch10_binary/benchmarks  5.939s
```

As expected, the single function decoding takes longer than with gob, but the performance is fast when you manually decode the data into structs. It's a bit more tedious to code, but the performance is there if you need it.

CHAPTER 11

Date and Time Recipes

11.0 Introduction

Time (and as an extension of that, date) manipulation is an important part of any programming language. We use it to keep track of time, calculate the elapsed duration between times, format dates and times for representation, and much more. However, time manipulation is often not as straightforward as it seems.

Our computers have two different types of clocks—a wall clock and a monotonic clock. The wall clock is what we're used to and is literally a *wall clock* that we look at to determine the time of day. The wall clock is usually synchronized with a Network Time Protocol (NTP) server to set the correct time on the computer. As a result, the wall clock can sometimes jump back and forth.

Also, the clock itself might be adjusted by a user or another program, so if you're using the wall clock to measure duration, the timing might not be accurate. For example, if the clock is changed while you're trying to measure how long a task takes, the results might become a negative number!

A monotonic clock, on the other hand, provides a time that is always going forward. With the same example, you won't be affected by clock changes or adjustments. In general, you should be using the wall clock to tell the time, and the monotonic clock to measure duration.

Like many programming languages, Go has a pretty good package to help with time and it's called (no surprise here) the `time` package.

11.1 Telling Time

Problem

You want to know the current time.

Solution

Use `time.Now` to return the current time.

Discussion

To find the current time, you can do this:

```
time.Now()
```

This will return a `Time` struct instance that represents both the wall clock and monotonic clock reading.

11.2 Doing Arithmetic with Time

Problem

You want to do simple addition and subtraction with time.

Solution

Use `Add` to add or subtract a duration to or from a given time. Use `Sub` to find the difference between two `Time` structs.

Discussion

A `Time` struct represents an instant in time with nanosecond precision. You can perform operations on it. For example, if you add a `Duration` to a `Time` struct, you will get another `Time` struct:

```
t0 := time.Now()
t1 := t0.Add(10 * time.Minute) // add 10 minutes
```

When you add 10 minutes to the `Time` struct, you get a new `Time` struct that is 10 minutes after the current time. What if you want to subtract 10 minutes? You might think you use the `Sub` method but no; actually you just `Add` a negative number instead:

```
t2 := to.Add(-10 * time.Minute) // subtract 10 minutes
```

So when do you use the `Sub` method? When you are trying to find the difference between two `Time` structs:

```
t3 := t1.Sub(t2)
```

This will return a `Duration` type that is just an `int64`. You'll learn more about `Duration` in Recipe 11.5.

11.3 Representing Dates

Problem

You want to represent a date.

Solution

Use `time.Time` because there is no separate struct for date in the Go standard library.

Discussion

There is no `Date` struct in the `time` package or anywhere in the standard library. There is, however, a `Date` function in the `time` package, which is used to create a specific date and time, and returns a `Time` struct:

```
t := time.Date(2009, time.November, 10, 23, 0, 0, 0, time.UTC)
```

The parameters you need are `year`, `month`, `day`, `hour`, `minute`, `second`, `nanosecond`, and `Location`. You need a non-nil value for the `Location` parameter (using UTC here) or else `Date` will panic.

You can see that all the values are `int` other than `Location`. `Month` is a special one because there is a `Month` type (which is an `int`), and you can extract the name of the month using a `String` method:

```
m := t.Month() // returns a Month type
m.String() // "November"
```

You can also extract the name of the day if you use the `Weekday` method on the `Time` struct:

```
w := t.Weekday() // returns a Weekday type
w.String() // "Tuesday"
```

11.4 Representing Time Zones

Problem

You want to include the time zone information in a `Time` struct.

Solution

The `Time` struct includes a `Location`, which is the representation of the time zone.

Discussion

A time zone is an area that follows a standard time that roughly follows longitude but in practice tends to follow political boundaries. All time zones are defined as offsets of the Coordinated Universal Time (UTC), ranging from UTC-12:00 to UTC+14:00.

The `Location` struct in the `time` package represents a time zone. Go's `time` package, like many other libraries in different programming languages, uses the time zone database managed by the Internet Assigned Numbers Authority (IANA).

This database, also known as *tz* or *zoneinfo*, contains data for many locations around the world, and the latest as of this writing is 28 March 2023. The naming convention for time zones in the `tz` database is in the form of `Area/Location`, for example `Asia/Singapore` or `America/New_York`.

There are a few ways to create a `Location` struct. First, you can use `LoadLocation` (loading the location from the `tz` database):

```
func main() {
    location, err := time.LoadLocation("Asia/Singapore")
    if err != nil {
        log.Println("Cannot load location:", err)
    }
    fmt.Println("location:", location)
    utcTime := time.Date(2009, time.November, 10, 23, 0, 0, 0, time.UTC)
    fmt.Println("UTC time:", utcTime)
    fmt.Println("equivalent in Singapore:", utcTime.In(location))
}
```

This is what you should see:

```
location: Asia/Singapore
UTC time: 2009-11-10 23:00:00 +0000 UTC
equivalent in Singapore: 2009-11-11 07:00:00 +0800 +08
```

You can see that the name of the time zone is just `08` instead of `Asia/Singapore`. `LoadLocation` simply loads the location from the `tz` database that's in the computer it's running on. If you want to use different data, you can use the `LoadLocationFromTZData` function instead.

Another way of creating a `Location` is to use the `FixedZone` function. This allows you to create any location you want (without being in the `tz` database) and also name it whatever you want:

```
func main() {
    location := time.FixedZone("Singapore Time", 8*60*60)
    fmt.Println("location:", location)
    utcTime := time.Date(2009, time.November, 10, 23, 0, 0, 0, time.UTC)
    fmt.Println("UTC time:", utcTime)
    fmt.Println("equivalent in Singapore:", utcTime.In(location))
}
```

This is what you should see:

```
location: Singapore Time
UTC time: 2009-11-10 23:00:00 +0000 UTC
equivalent in Singapore: 2009-11-11 07:00:00 +0800 Singapore Time
```

As you can see, the name of the time zone is now `Singapore Time`.

11.5 Representing Duration

Problem

You want to specify a duration of time.

Solution

Use the `Duration` type to represent a span of time.

Discussion

The main representation for a span of time in the `time` package is, of course, `Duration`. `Duration` is nothing fancy, though; it's just an `int64` with some interesting methods. You normally create a `Duration` like this:

```
d := 2 * time.Hour
```

This creates two hours. You can find the pretty form by calling the `String` method, or if you simply want to print it out to the screen, just do this:

```
fmt.Println(d)
```

and you should see this:

```
2h0m0s
```

What if you want to create 2 hours, 34 minutes, and 5 seconds? Just add them together, of course (since `Duration` is just an `int64`):

```
d := (2 * time.Hour) + (34 * time.Minute) + (5 * time.Second)
```

If you want to find out the equivalent in minutes, you can do this:

```
d.Minutes()
```

Or in seconds, milliseconds, and so on using the respective methods. However, there is no equivalent for anything larger than `time.Hour`.

11.6 Pausing for a Specific Duration

Problem

You want to pause the program for a duration of time.

Solution

Use `time.Sleep` to pause for a given duration.

Discussion

Sometimes you simply want to pause the program (or any goroutine) for some time. Whether it is because you want to wait for something to happen or to simulate processing time, it's a useful trick to have up your sleeve.

To pause for a given duration, use `time.Sleep` and pass in the duration:

```
time.Sleep(2 * time.Minutes) // pause for 2 minutes
```

This will pause the current goroutine for two minutes. Since the main program is a goroutine, if you run this in the main program, it will pause for two minutes while other goroutines can continue running.

11.7 Measuring Lapsed Time

Problem

You want to measure the lapsed time and make sure that it is accurate.

Solution

Use the monotonic clock in the `Time` struct to find the lapsed time.

Discussion

One of the more popular uses of the `time` package, or any time-related libraries, is to measure elapsed time. Go, like many other programming languages, uses the monotonic clock in the computer for this. Unlike some other programming languages,

though, the standard library in Go uses the same `time` package, and even methods, for both purposes.

But where do you see this monotonic time? The `Time` struct contains the data but only under certain circumstances. If you create a `Time` struct instance using `time.Now`, and print out the struct, you will be able to see the monotonic clock:

```
t := time.Now()
fmt.Println(t)
```

When you run it you see this:

```
2021-10-09 13:10:43.311791 +0800 +08 m=+0.000093742
```

The `m=+0.000093742` part is the monotonic clock. The part before that is the wall clock. This is what the package documentation says:

> If the time has a monotonic clock reading, the returned string includes a final field "m=±<value>", where value is the monotonic clock reading formatted as a decimal number of seconds.
>
> — `time` package documentation

But what does it mean by *the monotonic clock reading formatted as a decimal number of seconds*? It's a really small number! Actually, this just shows how long your program has been running. This recipe has been showing you just a snippet, but this snippet is part of a program. Take a look at the larger program:

```
package main
import (
    "fmt"
    "time"
)
func main() {
    t := time.Now()
    fmt.Println(t)
}
```

Run this and you should also see this:

```
2021-10-09 13:10:43.311791 +0800 +08 m=+0.000093742
```

You can change it a bit by extending the running time before you print out the current time:

```
func main() {
    time.Sleep(10 * time.Second) // pretend to do something for 10s
    t := time.Now()
    fmt.Println(t)
}
```

Here you're going to `Sleep` for 10 seconds. This is what you will see after 10 seconds:

```
2021-10-09 13:21:28.090604 +0800 +08 m=+10.000173581
```

Does this make more sense now? You can see that `m` is slightly more than 10 seconds because that's the running time of the program at that point in time. Now, here's how you measure lapsed time with the monotonic clock:

```
func main() {
    time.Sleep(10 * time.Second) // pretend to do something for 10s
    t1 := time.Now()
    t2 := time.Now()
    fmt.Println("t1:", t1)
    fmt.Println("t2:", t2)
    fmt.Println("difference:", t2.Sub(t1))
}
```

This is what you should see after 10 seconds:

```
t1: 2021-10-09 15:12:12.432516 +0800 +08 m=+10.005330678
t2: 2021-10-09 15:12:12.432516 +0800 +08 m=+10.005330984
difference: 306ns
```

If you subtract the two monotonic clock values, you see it will match to 306ns, which is the amount of time that elapsed between calling `time.Now` twice consecutively:

```
10.005330984 - 10.005330678 = 0.000000306
```

The interesting point to note here is that if you use the wall clock, you won't be able to tell the difference at all!

```
t1: 2021-10-09 15:12:12.432516 +0800 +08 m=+10.005330678
t2: 2021-10-09 15:12:12.432516 +0800 +08 m=+10.005330984
```

As noted earlier, the monotonic clock data is not always available in the `Time` struct. Methods such as `AddDate`, `Round`, and `Truncate` are wall clock computations so the `Time` structs they return won't have the monotonic clock. Similarly, `In`, `Local`, and `UTC` interpret the wall clock, so the `Time` structs they return won't have the monotonic clock either.

You can also remove the monotonic clock yourself. Just use the `Round` method with a 0 parameter:

```
t := time.Now().Round(0)
fmt.Println(t)
```

You should get something like this:

```
2021-10-09 15:25:31.369518 +0800 +08
```

You no longer see the monotonic clock here. So what happens when you try to call `Sub` (or any other monotonic methods) on a `Time` struct instance that doesn't have the monotonic clock?

```
func main() {
    t1 := time.Now().Round(0)
    t2 := time.Now().Round(0)
    fmt.Println("t1:", t1)
    fmt.Println("t2:", t2)
    fmt.Println("difference:", t2.Sub(t1))
}
```

Now `t1` and `t2` have their monotonic clocks stripped away. If you try to find the difference between these two times, you will get a big fat 0. This is because the operation will be done on the wall clock instead of the monotonic clock, and the wall clock just isn't fine-grained enough:

```
t1: 2021-10-09 15:28:38.451622 +0800 +08
t2: 2021-10-09 15:28:38.451622 +0800 +08
difference: 0s
```

You can pretend to do something in between the two calls to `time.Now`:

```
func main() {
    t1 := time.Now().Round(0)
    time.Sleep(10 * time.Second) // pretend to do something for 10s
    t2 := time.Now().Round(0)
    fmt.Println("t1:", t1)
    fmt.Println("t2:", t2)
    fmt.Println("difference:", t2.Sub(t1))
}
```

You should now be able to see the difference:

```
t1: 2021-10-09 15:25:31.369518 +0800 +08
t2: 2021-10-09 15:25:41.372606 +0800 +08
difference: 10.003088s
```

11.8 Formatting Time for Display

Problem

You want to format a `Time` struct instance to display in various formats.

Solution

Use the `Format` function with the appropriate layout pattern to display time in the format you want.

Discussion

Date and time formats are important—important enough that there is an ISO standard, ISO 8601, as well as an IETF RFC, the RFC 3339. Both these standards define a date and time format, with ISO 8601 being the wider-scoped document. Other than these two there are also a bunch of RFCs that talk about date and time formats. These time formats are described in the following paragraphs.

ISO 8601

ISO 8601 uses the Gregorian calendar and a 24-hour clock system. Calendar dates are commonly in the basic format YYYYMMDDD or YYMMDD or YYYY-MM-DD in the extended format. When the day is omitted, only the extended format YYYY-MM is allowed.

ISO 8601 allows for week date representations that look something like this: 2021-W04–2. The week date formats are YYYYWww or YYYYWwwD or YYYY-Www or YYYY-Www-D. The week is represented by the week number prefixed by a W, from W01 to W53. D is the weekday number from 1 (Monday) to 7 (Sunday). The basic format for time is Thhmmss with the extended format being Thh:mm:ss. The latest ISO 8601–1:2019—allows T to be omitted in the extended format.

Time zones in ISO 8601 are represented as UTC if there is a Z after the time without space (for example 13:57 UTC is 1:57 p.m. UTC time, represented in ISO 8601 as 13:57Z). If there is an offset after the time without space, it will be the time zone offset (for example 1:57 p.m. in Singapore is represented as 13:57+8:00). Any time representation with Z or offset is considered local time.

RFC 3339

The RFC 3339 is very similar to the ISO 8601 and is sometimes considered a subset of the ISO standard. However there are certain differences; for example, T is mandated for time representations in ISO 8601 before the 2019 version, all dates must contain hyphens in RFC 3339, and RFC 3339 doesn't cater to week date representations.

ISO standards are created by the industry and government and often used for regulations so the language is more formal and used more widely by many different industries. RFCs are standards created by the Internet Engineering Task Force (IETF) for the internet and other technologies so they tend to be more focused on the internet.

Other formats

Besides these two most widely used standards, there are other time and date standards, for example:

RFC 822
: RFC 822 is the standard for ARPA Internet Text Messages (what we know as email) and is one of the oldest and most important internet standards. It has nothing to do with date or time, but it does provide a date and time specification in the standard.

RFC 850
: RFC 850 is the standard for the Interchange of USENET Messages. Again it has nothing to do with date and time, but obviously, USENET messages need a date-time format, and it's defined here as well. USENET is a distributed discussion system that was very popular in the earlier days of the internet and the precursor to the internet forums.

RFC 1123
: RFC 1123 is part of a pair of standards that defines the requirements for internet hosts. This RFC described application and support protocols while the companion RFC 1122 covered the communication protocol layers. As before, it has a date and time specification, which also references RFC 822.

Formatting time

The `time` package formats time via pattern-based layouts. What this means is that you provide a particular format layout that is like a reference, and the `time` package will format the time accordingly.

It's quite straightforward to do this. The `Format` method of the `Time` struct will take in a layout and return the formatted string:

```
func main() {
 t := time.Now()
 fmt.Println(t.Format("3:04PM"))
 fmt.Println(t.Format("Jan 02, 2006"))
}
```

When you run this you should see something like:

```
1:45PM
Oct 23, 2021
```

That's simple enough. The `time` package makes it even simpler because it has several layout constants that you can use directly (note those RFCs described earlier):

```
func main() {
 t := time.Now()
 fmt.Println(t.Format(time.UnixDate))
 fmt.Println(t.Format(time.RFC822))
 fmt.Println(t.Format(time.RFC850))
 fmt.Println(t.Format(time.RFC1123))
 fmt.Println(t.Format(time.RFC3339))
}
```

Here's the output:

```
Sat Oct 23 15:05:37 +08 2021
22 Oct 23 15:05 +08
Saturday, 23-Oct-21 15:05:37 +08
Sat, 23 Oct 2021 15:05:37 +08
2021-10-23T15:05:37+08:00
```

Besides the RFCs, there are a few other formats including the interestingly named Kitchen layout, which is just 3:04 p.m. Also if you're interested in doing timestamps, there are a few timestamp layouts as well:

```
func main() {
 t := time.Now()
 fmt.Println(t.Format(time.Stamp))
 fmt.Println(t.Format(time.StampMilli))
 fmt.Println(t.Format(time.StampMicro))
 fmt.Println(t.Format(time.StampNano))
}
```

Here's the output:

```
Oct 23 15:10:53
Oct 23 15:10:53.899
Oct 23 15:10:53.899873
Oct 23 15:10:53.899873000
```

You might have seen earlier that the layout patterns are like this:

```
t.Format("3:04PM")
```

The full layout pattern is a layout by itself—it's the `time.Layout` constant:

```
const Layout = "01/02 03:04:05PM '06 -0700"
```

As you can see, the numerals are ascending, starting with 1 and ending with 7. Because of a *historic error* (mentioned in the `time` package documentation), the date uses the American convention of putting the numerical month before the day. This means 01/02 is January 2 and not 1 February.

The numbers are not arbitrary. Take this code fragment, where you use the format "`3:09pm`" instead of "`3:04pm`":

```
func main() {
 t := time.Date(2009, time.November, 10, 23, 45, 0, 0, time.UTC)
 fmt.Println(t.Format(time.Kitchen))
 fmt.Println(t.Format("3:04pm")) // the correct layout
 fmt.Println(t.Format("3:09pm")) // mucking around
}
```

This is the output:

```
11:45pm
11:45pm
11:09pm
```

You can see that the time is 11:45 p.m., but when you use the layout 3:09 p.m., the hour is displayed correctly while the minute is not. It shows :09, which means it's considering 09 as the label instead of a layout for minute.

What this means is that the numerals are not just a placeholder for show. The month must be 1, day must be 2, hour must be 3, minute must be 4, second must be 5, year must be 6, and the time zone must be 7.

This is something that new Go developers (or anyone being careless) can stumble on. The `Format` method doesn't return an error so you can't tell if the time format returned is correct, unless you have very explicit tests for them.

11.9 Parsing Time Displays Into Structs

Problem

You want to parse time display strings to `Time` structs.

Solution

Use the `Parse` method to convert a time display string to a `Time` struct.

Discussion

While the `Format` method converts a `Time` struct instance to a string representation according to the layout, the `Parse` method converts a string representation into a `Time` struct, according to a layout. The layout here is the same as in `Format`:

```
func main() {
 str := "4:31am +0800 on Oct 1, 2021"
 layout := "3:04pm -0700 on Jan 2, 2006"
 t, err := time.Parse(layout, str)
 if err != nil {
  log.Println("Cannot parse:", err)
 }
 fmt.Println(t.Format(time.RFC3339))
}
```

This is the result:

```
2021-10-01T04:31:00+08:00
```

You might notice that the `Parse` function returns two values, and one of them is an error. This is usually indicative that errors can occur during parsing. Sure enough, if your string or your layout is not what is expected, the compiler will scold you.

Say you have this layout pattern instead:

```
layout := "3:04pm -0700"
```

You are removing the date information because you just want to have the time. This is the error you will get:

```
2021/10/23 18:16:42 Cannot parse: parsing time "4:31am +0800 on Oct 1, 2021":
extra text: " on Oct 1, 2021"
```

Your value string has extra stuff and the compiler doesn't like it. What about the other way around; you keep the earlier layout but instead remove the date part of the value string like this:

```
str := "4:31am +0800"
```

You get an error as well. The compiler now complains that you don't have enough stuff in your value string (it's a bit sarcastic):

```
2021/10/23 18:19:17 Cannot parse: parsing time "4:31am +0800" as "3:04pm -0700 on
Jan 2, 2006": cannot parse "" as " on "
```

Earlier you saw that using the wrong numeral for the layout will result in wrong formatting. This happens in `Parse` as well, except that this time you will be getting an error. Say you change the layout to this:

```
layout := "3:09pm -0700 on Jan 2, 2006"
```

Notice that you changed the layout to :09 instead of :04. You will get this error:

```
2021/10/23 20:46:36 Cannot parse: parsing time "4:31am +0800 on Oct 1, 2021" as
"3:09pm -0700 on Jan 2, 2006": cannot parse "31am +0800 on Oct 1, 2021" as ":09"
```

If you're using a time zone abbreviation like SGT or EST, it will still be parsed and it will be considered a location. However, the offset will be zero. Yes, you got that right. It will say what you want it to say but totally ignore your intention of using it as the time zone.

Take a look at what this means. If your value string and layout are like this:

```
str := "4:31am SGT on Oct 1, 2021"
layout := "3:04pm MST on Jan 2, 2006"
```

Print the parsed `Time` struct instance with a few more layouts from the package:

```
fmt.Println(t.Format(time.RFC822))  // "02 Jan 06 15:04 MST"
fmt.Println(t.Format(time.RFC822Z)) // "02 Jan 06 15:04 -0700"
fmt.Println(t.Format(time.RFC3339)) // "2006-01-02T15:04:05Z07:00"
```

The RFC 822Z layout uses the numeric offset while the RFC 822 layout uses the abbreviation for time zone representation. This is what you get:

```
01 Oct 21 04:31 SGT
01 Oct 21 04:31 +0000
2021-10-01T04:31:00Z
```

As you can see, the abbreviation is printed nicely, and no error is returned but the offset is obviously wrong since SGT is +0800. In fact, in the RFC 3339 layout shows that it is actually in UTC (it shows a Z).

This is something that can easily trip up someone who is not aware or careless because there is an error returned.

Why is it like this? According to the package documentation:

> When parsing a time with a zone abbreviation like MST, if the zone abbreviation has a defined offset in the current location, then that offset is used. The zone abbreviation "UTC" is recognized as UTC regardless of location. If the zone abbreviation is unknown, Parse records the time as being in a fabricated location with the given zone abbreviation and a zero offset.
>
> — `time` package documentation

To solve this problem, you should use a numeric offset like +0800 instead of an abbreviation like SGT.

CHAPTER 12

Structs Recipes

12.0 Introduction

In Go, a *struct* is a collection of named data fields. It is used to group related data to represent an entity. Structs are usually defined and later instantiated when needed.

Structs are an important construct in Go. For programmers who come from an object-oriented programming background, this will be familiar (and maybe unfamiliar at the same time) as structs can be considered a stand-in for classes. Some concepts are similar; for example, you can define a struct like you define a class, and you can define methods for a struct as well. Go also supports polymorphism using interfaces and encapsulation using exported and nonexported struct fields and methods.

However, structs do not inherit from other structs (you can compose structs with other structs, though), and there are no objects (though instances of structs are sometimes called objects) because there are no classes.

> Go is a profoundly object oriented language.
>
> —Rob Pike

> Go is object oriented, but it's not type oriented.
>
> —Russ Cox

12.1 Defining Structs

Problem

You want to define a struct.

Solution

Define a struct using the `type … struct` syntax.

Discussion

To define a struct you can use the `type … struct` syntax. Start with a struct that represents data related to a person:

```
type Person struct {
        Id     int
        Name   string
        Email  string
}
```

You can group all kinds of data types within a struct, even other structs. For example, if you want to add a date of birth to the `Person` struct:

```
type Person struct {
        Id        int
        Name      string
        Email     string
        BirthDate time.Time
}
```

In this example, both the struct name and all the field names are capitalized (that is, they start with an uppercase letter). This means both the class and the fields are exported outside of the package. As you may have guessed, if the names are not capitalized, it means that the struct or the fields are not exported and not visible outside of the package.

If the field names are not capitalized but the struct name is, that means the struct is exported and visible outside of the package but the fields are not directly accessible:

```
type Person struct {
        Id        int
        Name      string
        Email     string
        birthDate time.Time
}
```

In this example, only `Id`, `Name`, and `Email` are visible outside of the package. This is useful because when you're creating a package you might want to put the data

together but you want the programmer using the package to only use the methods and not the struct fields directly. This is a quite common pattern.

You can even define struct fields that are functions. This is different from defining methods for a struct. Here's how that looks:

```
type NameFunc func(string, string) string

type Person struct {
    Id         int
    GivenName  string
    FamilyName string
    Email      string
    Name       NameFunc
}
asian := func(giveName string, familyName string) string {
    return familyName + " " + giveName
}

western := func(giveName string, familyName string) string {
    return giveName + " " + familyName
}
```

Asian names are often displayed with the family name first, followed by the given name, while Western names display the given name first, followed by the family name. However, under different conditions, the ordering of the names might be different as well. For example, Japanese names are usually in the Western name order when mentioned in Western media, while in Japanese (or other Asian) media their names are in the Asian name order. Hayao Miyazaki, the famous founder of Studio Ghibli, has the family name Miyazaki and the given name Hayao. In Western media, he is known as Hayao Miyazaki, while in Japanese media he is known as Miyazaki Hayao.

In this case, you need to display names dynamically, depending on the context. You can do this by making the `Name` field a function, and accordingly, during runtime, you can assign whichever function best fits the situation. In addition, this gives flexibility to some other programmers using the struct to define custom functions to display names accordingly:

```
person := Person{
    Id:         1,
    GivenName:  "Sau Sheong",
    FamilyName: "Chang",
    Email:      "sausheong@email.com",
    Name:       asian,
}

fmt.Println("Name:", person.Name(person.GivenName, person.FamilyName))
person.Name = western
fmt.Println("Name:", person.Name(person.GivenName, person.FamilyName))
```

This is what you will see when you run the code:

```
Name: Chang Sau Sheong
Name: Sau Sheong Chang
```

So how does this differ from using methods? Methods are associated with the structs and cannot be changed at runtime, but a function field can be reassigned during runtime. Function fields are also often used to store callback functions.

12.2 Creating Struct Methods

Problem

You want to create a method for your struct.

Solution

Create a function that associates with a struct.

Discussion

You can define methods, which are functions that are associated with the struct. Here's how you can associate a function with a struct.

First, you add a new field called `roles` to the `Person` struct you used earlier. This is a slice of strings that stores the roles that the person plays.

Then you make the data field nonexported so that it's not directly accessible outside of the package. However, you still want the roles to be available (just not modifiable) so you add an exported method called `Roles` that returns a slice of strings that represents the roles:

```
type Person struct {
        Id         int
        GivenName  string
        FamilyName string
        Email      string
        Name       NameFunc
        roles      []string
}

func (person Person) Roles() (roles []string) {
        roles = make([]string, len(person.roles))
        copy(roles, person.roles)
        return
}
```

The slice is a copy of the actual roles because slices are references; if you return it directly, the caller can still modify the roles even though it's not exported.

To check if a person has a specific role, use a method called `HasRole` that takes in a role to check and returns either `true` or `false` depending on if the role is found in the `roles` field:

```
func (person Person) HasRole(role string) (has bool) {
        for _, r := range person.roles {
                if role == r {
                        has = true
                }
        }
        return
}
```

You can tell the difference between a function and a method by what is placed after the `func` keyword. Functions have the function name after the `func` keyword while methods have a *receiver* after the `func` keyword and before the function name. A receiver is a copy of the struct and can be used within the method. In this example, `(person Person)` is placed after the `func` keyword, where `person` is the receiver of type `Person`. It's important to know that the receiver is a copy of the struct and not the actual struct, and whatever you do with it will not be reflected in the original.

If you want the changes made within the method to affect the struct, you will need to make the receiver a pointer. Take a look at another method:

```
func (person *Person) AddRole(role string) {
        person.roles = append(person.roles, role)
}
```

The receiver `person` is now a pointer to the struct `Person`, and anything you do with the receiver will be done on the actual struct. In this case, when you append a new role to existing roles, it will be added to the struct.

Here's how you can use these methods:

```
person := Person{
    Id:         1,
    GivenName:  "Sau Sheong",
    FamilyName: "Chang",
    Email:      "sausheong@email.com",
}
fmt.Println("Roles:", person.Roles())
person.AddRole("approver")
fmt.Println("Roles:", person.Roles())
fmt.Println("Has approver role?", person.HasRole("approver"))
```

This is what you should see:

```
Roles: []
Roles: [approver]
Has approver role? true
```

First, you create a `Person` struct instance called `person`. You check on the roles that this person has by calling the `Roles` method, using the dot notation. This notation is just putting a dot after the struct instance, followed by the method name: very similar to the way you call functions. The difference is when you call functions, you put a dot after the *package name* followed by the function name.

As you can see, there are no roles in the `person` struct instance. You call the `AddRole` method on the `Person` struct to add a new role to it. After doing that, you call `Roles` again to show the roles in the `person` struct instance. You see the changes in the roles, precisely because the `AddRole` method made changes to the struct instance itself. If the receiver for `AddRole` hadn't been a pointer to a struct, but instead was just a struct, the original struct would not be modified and you would not see the new role.

While it is common to assign methods to structs, you can also create methods on any user-defined types, even if you redefine an existing type:

```
type Strings string

func (s Strings) Len() int { return len(s) }
```

In this example, you redefined the `string` type as `Strings` and added a method called `Len` that returns the length of the string. You can now create a string using the `Strings` type and then call `Len` on an instance of `Strings` to get its length:

```
s := Strings("hello")
fmt.Println(s.Len())
```

12.3 Creating and Using Interfaces

Problem

You want to create interfaces for code reusability and better code organization.

Solution

Use the `type … interface` syntax to create an interface.

Discussion

Whether Go is an object-oriented programming language is quite gray. On one hand, it doesn't have classes or type hierarchy or inheritance, but on the other, it supports several object-oriented programming concepts like encapsulation and composition and what we'll be discussing in this recipe—polymorphism.

Polymorphism is a core concept in object-oriented programming. In simple terms, polymorphism is a concept that objects of different types can be accessed through a single interface. If you imagine a circular hole of a certain radius, you can pass a ball

through it, or you can pass a coin through it, or even a cylinder. Each of these objects is a different type but they all fit in the same circular hole interface. Go implements polymorphism through interfaces. An interface in Go is a type that has a certain set of methods. Any types that implement the same methods are considered to be of that interface.

Here's an example from the standard library:

```
type Reader interface {
        Read(p []byte) (n int, err error)
}
```

The `Reader` interface from the `io` package has only one method, which is `Read`, a method that takes in a slice of bytes and returns an integer and an error. Any struct that has a method named `Read` with the same method signature will be considered to implement the `Reader` interface.

You can think of it this way: any struct that implements `Read` with the same method signature is a `Reader`. By convention, one-method interfaces have noun names, following the verb names for the methods. For example, the interface that has a single verb method `Read` would be called `Reader`. Other names like `Writer`, `Stringer`, `Formatter`, and so on, have single methods named `Write`, `String`, `Format`, and so on.

This example shows how you can use interfaces. You'll use the `Person` struct again:

```
type Person struct {
    Id        int
    Email      string
}
```

Create a new interface named `Worker` with only one method, `Work`:

```
type Worker interface {
        Work()
}
```

You want to pay your workers, so you create a function named `Pay` that has a single parameter of type `Worker`:

```
func Pay(worker Worker) {
        worker.Work()
        fmt.Println("and getting paid!")
}
```

Now, try to create a new `Person` struct instance and try to pay this person:

```
person := Person{
    Id:    1,
    Email: "sausheong@email.com",
}

Pay(person)
```

When you try to compile this, Go will quickly tell you that `person` is not a `Worker` because it doesn't implement the `Work` method:

```
cannot use person (type Person) as type Worker in argument to Pay:
        Person does not implement Worker (missing Work method)
```

Once `Person` implements `Work`, you can compile the code:

```
func (person Person) Work() {
        fmt.Print("Working hard ... ")
}
```

This is what you see when you run the program:

```
Working hard ... and getting paid!
```

The `Pay` function only accepts types that implement the `Work` method, which is why you can confidently call the `Work` method on the worker. This also means you can pass other structs into the `Pay` method as long as it implements `Work`.

Do that and see what happens:

```
type Machine struct {
        Id        int
        IPAddress string
}

func (machine Machine) Work() {
        fmt.Print("Automating stuff ... ")
}
```

Now try to pay the machine:

```
person := Person{
    Id:    1,
    Email: "sausheong@email.com",
}
machine := Machine{
    Id:        2,
    IPAddress: "192.168.0.1",
}

Pay(person)
Pay(machine)
```

This is what you should see:

```
Working hard ... and getting paid!
Automating stuff ... and getting paid!
```

In other words, it doesn't matter if it's a `Person` or a `Machine`; if it works it will get paid.

You might notice that no special syntax indicates that a struct implements a particular interface, unlike some other languages. This could make it difficult to know which interfaces your struct implements.

12.4 Creating Struct Instances

Problem

You want to create an instance of a struct.

Solution

Create a struct instance directly using the name of the struct, or a pointer to a struct instance using the `new` keyword.

Discussion

There are two ways to create an instance of a struct. The first is to directly use the name of the struct:

```
type Person struct {
        Id     int
        Name   string
        Email  string
}

person := Person{}
```

This creates an empty struct instance, which you can populate subsequently by accessing the struct fields:

```
person.Id = 1
person.Name = "Chang Sau Sheong"
person.Email = "sausheong@email.com"
```

A quicker way is to create and initialize the struct instance at the same time:

```
person := Person{1, "Chang Sau Sheong", "sausheong@email.com"}
```

This is pretty straightforward. However, you need to specify all field values.

You can also make the code clearer by indicating the name of the data field during initialization. When you do this, you can leave out field values:

```
person := Person{
    Id:    1,
    Email: "sausheong@email.com",
}
```

Note that the last data field line also ends with a comma.

To create a reference to a struct instance you can use the address operator (`&`):

```
person := &Person{
    Id:    1,
    Name:  "Chang Sau Sheong",
    Email: "sausheong@email.com",
}
```

This brings us to the second way of creating struct instances, which is to use the `new` built-in function. The `new` function is used for more than creating struct instances though; it can be used to create references to a specified type. In this case, you're using it to create a reference to a struct instance:

```
person := new(Person)
```

When creating a struct instance this way, `new` returns a reference to the struct instance and not the actual instance itself. In other words, this is equivalent to the following code:

```
person := &Person{}
```

Should you pass a struct instance by copy or by reference? The usage is quite clear; you would want to pass by reference if you want the function to modify the struct instance in some way. If not, both ways work. In this case, wouldn't it be better to always pass by reference so you don't need to think too much about it?

Not really, because there is a difference in performance. There are two ways we normally pass struct instances around. The first is passing a struct instance down to a function either by copy or by reference.

Here's a quick benchmark on the performance:

```
func byCopyDown(p Person) {
        _ = fmt.Sprintf("%v", p)
}

func byReferenceDown(p *Person) {
        _ = fmt.Sprintf("%v", p)
}
```

You create two functions that take a struct instance by copy and a struct instance by reference and use them the same way. Then you create two benchmark tests to see the difference in performance:

```
func BenchmarkStructInstanceDownCopy(b *testing.B) {
        for i := 0; i < b.N; i++ {
                p := Person{
                        Id:         1,
                        GivenName:  "Sau Sheong",
                        FamilyName: "Chang",
                        Email:      "sausheong@email.com",
                }
```

```
                byCopyDown(p)
        }
}

func BenchmarkStructInstanceDownReference(b *testing.B) {
        for i := 0; i < b.N; i++ {
                p := &Person{
                        Id:         1,
                        GivenName:  "Sau Sheong",
                        FamilyName: "Chang",
                        Email:      "sausheong@email.com",
                }
                byReferenceDown(p)
        }
}
```

The benchmarks are straightforward; you just create the structs and pass them by copy or by reference into your earlier functions. When you run the benchmark, this is what you should see:

```
% go test -bench=BenchmarkStructInstanceDown
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch13_structs
BenchmarkStructInstanceDownCopy-10              2387568             490.7 ns/op
BenchmarkStructInstanceDownReference-10         1704202             703.6 ns/op
```

Passing down by copy is faster than passing down by reference. Now let's look at the second direction, which is passing the struct instance by copy or by reference back from the function to the caller.

You create two functions. One returns a struct instance by copy, and the other returns a struct instance by reference:

```
func byCopyUp() Person {
        return Person{
                Id:         1,
                GivenName:  "Sau Sheong",
                FamilyName: "Chang",
                Email:      "sausheong@email.com",
        }
}

func byReferenceUp() *Person {
        return &Person{
                Id:         1,
                GivenName:  "Sau Sheong",
                FamilyName: "Chang",
                Email:      "sausheong@email.com",
        }
}
```

As before, you create two benchmark tests to check their performance:

```
func BenchmarkStructInstanceUpCopy(b *testing.B) {
        var p Person
        b.ResetTimer()
        for i := 0; i < b.N; i++ {
                p = byCopyUp()
        }
        b.StopTimer()
        _ = fmt.Sprintf("%v", p)
}

func BenchmarkStructInstanceUpReference(b *testing.B) {
        var p *Person
        b.ResetTimer()
        for i := 0; i < b.N; i++ {
                p = byReferenceUp()
        }
        b.StopTimer()
        _ = fmt.Sprintf("%v", p)
}
```

When you run it this is what you should see:

```
% go test -bench=BenchmarkStructInstanceUp
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch13_structs
BenchmarkStructInstanceUpCopy-10                144514798            8.181 ns/op
BenchmarkStructInstanceUpReference-10           36263088             31.82 ns/op
PASS
ok      github.com/sausheong/gocookbook/ch13_structs    3.358s
```

The difference is even more significant when passing up. In general, if you don't need to pass structs around by reference, you should stick with passing by copy.

12.5 Creating One-Time Structs

Problem

You want to organize a collection of struct fields without defining it as a struct.

Solution

Use anonymous structs, which are structs that are not named and are created for one-time use.

Discussion

Go allows you to create anonymous structs, which are structs that are not named and do not need to be predefined:

```
person := struct {
    Id    int
    Name  string
    Email string
}{1, "Chang Sau Sheong", "sausheong@email.com"}
```

When you define an anonymous struct you have to create the struct instance at the same time. You can also create the struct instance with the struct field names:

```
person = struct {
    Id    int
    Name  string
    Email string
}{
    Id:    1,
    Name:  "Chang Sau Sheong",
    Email: "sausheong@email.com",
}
```

Either way, this is a one-time use; each time you want to create an instance of the struct you need to include the definition again. If you are going to use this struct more than once, you want to define it first.

There are a few use cases for anonymous structs but one particular case is pretty useful for bundling multiple pieces of unrelated data together to be sent to a function. Here's a concrete example.

Say you want to send a struct to an HTML template such that an HTML page will be generated by substituting data from the struct into various locations in the template:

```
type Person struct {
    Id    int
    Name  string
    Email string
}
person := Person{1, "Chang Sau Sheong", "sausheong@email.com"}
tpl := `<div>
    <div>{{ .Id }}</div>
    <div>{{ .Name }}</div>
    <div>{{ .Email }}</div>
</div>
`
templ := template.Must(template.New("person").Parse(tpl))
templ.Execute(os.Stdout, person)
```

You have a template named `tpl` and want to substitute various pieces of data in it from a struct you pass to it. In this case, you pass the `Person` struct to the `Execute`

method. The template will substitute the data at the various locations and if you run the preceding code, you will get this:

```
<div>
        <div>1</div>
        <div>Chang Sau Sheong</div>
        <div>sausheong@email.com</div>
</div>
```

This is all well and good but what if you want to display a piece of information on the same page that has nothing to do with a person, for example, a message for the user? You can't add a field into the `Person` struct, as that would be silly and awkward. You could define another struct that wraps around the message and the `Person` struct but it's only going to be used for that one single page. You have many other pages with other things to show besides the main piece of data. You can't possibly define multiple structs, each to pass in some data to the page.

Here is where anonymous structs make a lot of sense:

```
type Person struct {
    Id    int
    Name  string
    Email string
}
person := Person{1, "Chang Sau Sheong", "sausheong@email.com"}
tpl := `<h1>{{ .Message }}</h1>
<div>
        <div>{{ .P.Id }}</div>
        <div>{{ .P.Name }}</div>
        <div>{{ .P.Email }}</div>
</div>
`
data := struct {
    P       Person
    Message string
}{person, "Hello World!"}

templ := template.Must(template.New("person").Parse(tpl))
templ.Execute(os.Stdout, data)
```

Instead of sending in the `Person` struct, you create an anonymous struct that wraps around `Person` as well as a string message and send that in instead. This is what you will see when you run the code:

```
<h1>Hello World!</h1>
<div>
        <div>1</div>
        <div>Chang Sau Sheong</div>
        <div>sausheong@email.com</div>
</div>
```

This way, you can preserve your defined structs and, if there are other pieces of data to send to the page, you can create an anonymous struct to wrap around the data and send that in.

A caveat on using anonymous structs: when you pass anonymous structs to a function, that function needs to either accept `any` (as in the case of the `Execute` method earlier) or the fields need to match in that function.

12.6 Composing Structs from Other Structs

Problem

You want a struct that has data and methods of another struct.

Solution

Embed an unnamed struct within another struct. The outer struct will gain the data and methods of the inner struct.

Discussion

Inheritance is one of the major concepts in object-oriented programming. It allows subclasses (or child classes) to gain data and methods from the superclasses (or parent classes) it inherits from. For example, a `Square` subclass inherits its data and methods from its `Shape` superclass.

Inheritance is useful because it allows you to reuse code and break your models into submodels that can be extended and worked on independently. For example, you can put common data and methods in `Shape` and have more specialized data and functions in `Square`. You can also independently work on `Triangle` with its data and functions, and if you change `Shape` you can extend its capabilities to both `Square` and `Triangle`, but if you change `Triangle` it will not impact `Square`.

However, inheritance is not the only mechanism you can use for these purposes. Another popular mechanism is composition, which allows you to assemble your models from other models instead of inheriting from a superclass. Go implements composition by embedding an unnamed struct within another struct.

Structs can contain named data fields including structs, as described in Recipe 12.1. However, structs can also contain unnamed data fields, including structs. Whenever a struct has an unnamed data field that is a struct, the data fields within that inner struct are promoted to the outer struct:

```
type Person struct {
    Id    int
    Name  string
    Email string
}

type Manager struct {
    Person
}

mgr := Manager{}
mgr.Id = 2
mgr.Name = "Wong Fei Hung"
mgr.Email = "feihung@email.com"
```

In this example, `Manager` has a single unnamed data field that is of the type `Person`. The data fields of `Person` are promoted to `mgr` and you can access them directly, without referencing `Person` at all.

If you try to put `mgr` through to `fmt.Println`, you will see this:

```
{{2 Wong Fei Hung feihung@email.com}}
```

Notice the double braces `{{}}` used. This tells you that there is an inner struct instance.

You set the data fields after you created the struct instance. How can you initialize the struct instance as you create it?

```
mgr = Manager{
    Person{
        Id:    2,
        Name:  "Wong Fei Hung",
        Email: "feihung@email.com",
    },
}
```

You have to create a `Person` instance first when initializing it. Let's give the `Manager` struct another data field:

```
type Manager struct {
    Person
    Department string
}

mgr := Manager{}
mgr.Id = 2
mgr.Name = "Wong Fei Hung"
mgr.Email = "feihung@email.com"
mgr.Department = "Poh Chi Lam"
```

When you print out `mgr` again, you will see that the department is printed outside of the inner set of braces:

```
{{2 Wong Fei Hung feihung@email.com} Poh Chi Lam}
```

How about initializing the struct instance?

```
mgr = Manager{
    Person: Person{
        Id:    2,
        Name:  "Wong Fei Hung",
        Email: "feihung@email.com",
    },
    Department: "Poh Chi Lam",
}
```

You have to use a name field with the same name as the struct.

The same rules for exported fields apply here as well—capitalized variable names indicate the fields are exported, while lowercase variable names indicate they are not.

How about methods? They are also promoted to the outer struct:

```
func (person Person) Work() {
        fmt.Print("Working hard ... ")
}

mgr.Work()
```

You can call the `Work` method directly on `mgr` because the methods associated with `Person` are promoted to be available on `Manager` as well.

This also means any interfaces that `Person` satisfies, `Manager` also satisfies:

```
type Worker interface {
        Work()
}

func Pay(worker Worker) {
        worker.Work()
        fmt.Println("and getting paid!")
}

Pay(mgr)
```

Since `Person` satisfies the `Worker` interface, this means `Manager` also satisfies the `Worker` interface and therefore you can pass an instance of `Manager` to the `Pay` function. If you run the preceding code, you should get:

```
Working hard ... and getting paid!
```

12.7 Defining Metadata for Struct Fields

Problem

You want to define metadata to describe the struct fields.

Solution

Use struct tags to define metadata and the `reflect` package to access the tags.

Discussion

Sometimes you want to provide a bit more information about the data fields in structs, besides their name and type. You want to do this so you can process the structs better. Go provides a mechanism called *struct tags* that allows you to add string literals after each field to provide the information.

One very common place you find this is in the `json` package:

```
type Person struct {
    Id         int    `json:"id"`
    GivenName  string `json:"given_name"`
    FamilyName string `json:"family_name"`
    Email      string `json:"email"`
}
```

Go determines the mapping between the struct fields and the JSON elements using these struct tags. They are optional, but it is useful to do mapping if the names differ. There are other uses for JSON struct tags as well; for example, to omit the field from the JSON that is generated if it's empty, and others.

Many packages that process structs use struct tags. For example, `xml` package, the `protobuf` package, and the `sqlx` package all use struct tags. You can also roll out your support for struct tags using the `reflect` package.

Struct tags are defined in name-value pairs within the string literal. In the preceding example, where the string literal is `json:"given_name"`, the name is `json` and the value is a string `"given_name"`.

Here's how you can extract the values:

```
person := Person{
    Id:         1,
    GivenName:  "Sau Sheong",
    FamilyName: "Chang",
    Email:      "sausheong@email.com",
}

p := reflect.TypeOf(person)
```

```
for i := 0; i < p.NumField(); i++ {
    field := p.Field(i)
    fmt.Println(field.Tag.Get("json"))
}
```

To get the values from the struct tags, you use the `reflect` package. Start with getting the `Type` of the `person` variable, which is a struct. The `NumField` method on the `Type` tells you how many fields are in this `Type`, which is 4. Then you call the `Field` method on the `Type` with a given index to get the `StructField`. Call the `Tag` method on the `StructField` to get the `StructTag`, and finally call the `Get` method with the name to get the value.

If you run the code, this is what you should see:

```
id
given_name
family_name
email
```

Each struct tag can contain more than a single name-value pair, so you can call `Get` on different names to get to the corresponding values.

CHAPTER 13

Data Structure Recipes

13.0 Introduction

Go has four basic types of data structures: arrays, slices, maps, and structs. Chapter 12 discussed structs and this chapter will cover arrays, slices, and maps. Following is some background information on these data structures before getting into specific recipes for using them.

Arrays

Arrays are data structures that represent an ordered sequence of elements of the same type. Array sizes are static; they are set when the array is defined and cannot be changed subsequently. In Go, arrays are values. This is an important difference, because in some languages an array is a pointer to the first item in the array. This means if you pass an array to a function you will be passing a copy of the array, and this could be expensive.

Slices

Slices are data structures that also represent an ordered sequence of elements. Slices are built on top of arrays and are used much more often than arrays because of their flexibility. Slices have no fixed length. Internally, a slice is a struct that consists of a pointer to an array, the length of the segment of the array, and the capacity of the underlying array.

Maps

Maps are data structures that associate the values of one type (called the *key*) with values of another type (called the *value*). Such data structures are very common in

many other programming languages, called by different names like hash table, hash map, and dictionary. Internally, a map is a pointer to `runtime.hmap` structure.

It's important to understand that these three data structures are the basic building blocks of all other data structures, but they are also fundamentally very different from each other. In short, arrays are fixed-length order lists and are values. Slices are structs whose first element is a pointer to an array. Maps are pointers to an internal hashmap struct.

13.1 Creating Arrays or Slices

Problem

You want to create arrays or slices.

Solution

There are many ways to create arrays or slices including directly from literals, from another array, or using `make`.

Discussion

Arrays and slices are very different constructs, but conceptually they are very similar. As a result, creating arrays and slices is also very similar.

Defining arrays

You can define an array by declaring the size of the array in square brackets, followed by the data type of the elements. Arrays and slices can only have elements of the same type. You can also initialize the array during the declaration by putting the elements in curly brackets:

```
var numbers [10]int
fmt.Println(numbers)
rhyme := [4]string{"twinkle", "twinkle", "little", "star"}
fmt.Println(rhyme)
```

If you run the preceding code snippet, this is what you will see:

```
[0 0 0 0 0 0 0 0 0 0]
[twinkle twinkle little star]
```

The default value for an int or float array is 0. Note that the size of the array *cannot* be changed once it's created, but the elements can be changed. This makes arrays less flexible and is the main reason why slices are used more often than arrays.

Defining slices

Slices are constructs that are built on top of arrays. Most of the time when you need to deal with ordered lists, you would normally use slices because they are more flexible and also much cheaper to use, especially if the underlying array is large.

Slices are defined the same way, except you don't provide the size of the slice:

```
var integers []int
fmt.Println(integers)
var sheep = []string{"baa", "baa", "black", "sheep"}
fmt.Println(sheep)
```

If you run the preceding code snippet, this is what you will see:

```
[]
[baa baa black sheep]
```

Notice the difference here—a newly defined array must have a length and if it is not initialized, it will be filled with the default value. A newly defined slice, however, can have no elements and can be zero-length.

You can also create slices through the `make` function:

```
var integers = make([]int, 10)
fmt.Println(integers)
```

If you use `make`, you need to provide the type, the length, and an optional capacity. If you don't provide the capacity, it will default to the given length. This is what you will see if you run the preceding snippet:

```
[0 0 0 0 0 0 0 0 0 0]
```

As you can see, `make` initializes the slice as well.

To find out the length of an array or a slice, you can use the `len` function. To find out the capacity of an array or a slice, you can use the `cap` function. So what's the difference between a slice's length and its capacity? The length of a slice is the number of elements in it. The capacity of a slice is the length of the slice's underlying array, that is, the number of elements in the underlying array:

```
integers = make([]int, 10, 15)
fmt.Println(integers)
fmt.Println("length:", len(integers))
fmt.Println("capacity:", cap(integers))
```

Here, the `make` function allocates an array of 15 integers, then creates a slice with length of 10 and capacity of 15 that points at the first 10 elements of the array.

If you run the code, this is what you get:

```
[0 0 0 0 0 0 0 0 0 0]
length: 10
capacity: 15
```

You can also create new slices with the `new` method:

```
var ints *[]int = new([]int)
fmt.Println(ints)
```

The `new` method doesn't return the slice directly; it returns only a pointer to the slice. It also doesn't initialize the slice; instead it just zeros it. This is what you get when you run the code:

```
&[]
```

You can't create new arrays using the `make` function, but you can create new arrays using the `new` function:

```
var ints *[10]int = new([10]int)
fmt.Println(ints)
```

What you get is a pointer to an array:

```
&[0 0 0 0 0 0 0 0 0 0]
```

13.2 Accessing Arrays or Slices

Problem

You want to access elements in an array of a slice.

Solution

There are a few ways to access elements in an array or a slice. Arrays and slices are ordered lists, so elements in them can be accessed by their index. The elements can be accessed through a single index or a range of indices. You can also access them by iterating through the elements.

Discussion

Accessing arrays and slices are almost the same. As they are ordered lists, you can access an element of an array or a slice through its index:

```
numbers := []int{3, 14, 159, 26, 53, 59}
```

In the preceding slice, the 4th element, given the index 3 (we start with 0) is 25, and can be accessed using the name of the variable, followed by square brackets, and indicating the index within the square brackets:

```
numbers[3] // 26
```

You can also access a range of numbers by using the starting index, followed by a colon (`:`) and the ending index. The ending index is not included and this results in a slice (of course):

```
numbers[2:4] // [159, 26]
```

If you don't have a starting index, the slice will start at 0:

```
numbers[:4] // [3 14 159 26]
```

If you don't have an ending index, the slice will end with the last element of the original slice (or array):

```
numbers[2:] // [159 265 53 59]
```

Needless to say, if you don't have either a starting or ending index, the whole original slice is returned. While this sounds silly, there is a valid use for this—if you use this on an array, it simply converts the array to a slice:

```
numbers := [6]int{3, 14, 159, 26, 53, 59} // an array
numbers[:] // this is a slice
```

In this code snippet, `numbers` is an array. When you index `numbers`, you will get a slice.

You can also access elements in an array or a slice by iterating through the array or slice:

```
for i := 0; i < len(numbers); i++ {
    fmt.Println(numbers[i])
}
```

This uses a normal `for` loop, iterating through the length of the slice, and incrementing the count at every loop. The resulting output is as follows:

```
3
14
159
25
53
59
```

This uses a `for ... range` loop and returns the index `i` and the value `v`:

```
for i, v := range numbers {
    fmt.Printf("i: %d, v: %d\n", i, v)
}
```

The resulting output is:

```
i: 0, v: 3
i: 1, v: 14
i: 2, v: 159
i: 3, v: 25
i: 4, v: 53
i: 5, v: 59
```

13.3 Modifying Arrays or Slices

Problem

You want to add, insert, or remove elements in an array or a slice.

Solution

There are a few ways to modify elements in an array or a slice. Elements can be appended to the end of the slice, inserted at a particular index, or removed or modified.

Discussion

Besides accessing the elements in an array or slice, you may also want to add, modify, or remove elements in a slice. While you cannot add or remove elements in an array, you can modify its elements. Arrays are *not* immutable; they just have fixed lengths so you can't shrink or expand them:

```
numbers := []int{3, 14, 159, 26, 53, 58}
numbers[2] = 1000
```

When you modify the element at the given index, it will change the array or slice accordingly. In this case, when you run the code, you will get this:

```
[3 14 1000 26 53 58 97]
```

Appending

Arrays cannot change their size, so appending or adding elements to an array is out of the question. Appending to slices is quite straightforward, though. You just use the `append` function, passing it the slice and the new element, and a new slice that has the appended element will be returned:

```
numbers := []int{3, 14, 159, 26, 53, 58}
numbers = append(numbers, 97)
fmt.Println(numbers)
```

If you run the preceding code, you will get this:

```
[3 14 159 26 53 58 97]
```

You cannot append an element of a different type to the slice. However, you can append multiple items to the slice:

```
numbers = append(numbers, 97, 932, 38, 4, 626)
```

This means you can append a slice (or an array) to another slice by using the *slice unpacking notation* (…):

```
nums := []int{97, 932, 38, 4, 626}
numbers = append(numbers, nums...)
```

However, appending both an element and an unpacked slice at the same time is not allowed. You can choose to append multiple elements or an unpacked slice, but not both at the same time:

```
numbers = append(numbers, 1, nums...) // this will produce an error
```

Inserting

While appending adds an element to the end of the slice, inserting means adding an element anywhere in between elements in a slice. Again, this only applies to slices because array sizes are fixed.

There is no built-in function for insertion, but you can still use `append` for the task. Let's say you want to insert the number 1000 between elements at index 2 and 3, which are ints 159 and 26, respectively:

```
numbers := []int{3, 14, 159, 26, 53, 58}
numbers = append(numbers[:2+1], numbers[2:]...)
numbers[3] = 1000
```

First, you need to create a slice from the start of the original slice to the index 2 plus 1 (you're adding just 1 element). This will reserve a space for the new element you want to add. Next, you append this slice to another slice that begins at index 2 to the end of the original slice, using the unpack notation. With this, you have created a new element between the index 2 and 3. After the append, this is what `numbers` looks like:

```
[3 14 159 159 26 53 58]
```

Notice that there are now seven elements in the slice; the new element is inserted between 159 and 26, with the value of 159. Finally, you set the new element, 1000, at index 3. As a result, you will get this new slice:

```
[3 14 159 1000 26 53 58]
```

What if you want to add an element to the beginning of the slice; for example, you want to add the integer 2000 to the beginning of the slice. This is quite simple; you simply append the value, in the form of a slice, to the unpacked values of the original slice:

```
numbers = append([]int{2000}, numbers...)
```

That was the case with inserting a single element. What if you want to add a slice of numbers in between two elements of another slice of numbers? For example, you want to insert the slice `[]int{1000, 2000, 3000, 4000}` in between index 2 and 3 of the `numbers` slice like before.

There are a few ways of doing this, but stick with using `append`, which is one of the shortest ways:

```
numbers = []int{3, 14, 159, 26, 53, 58}
inserted := []int{1000, 2000, 3000, 4000}

tail := append([]int{}, numbers[3:]...)
numbers = append(numbers[:3], inserted...)
numbers = append(numbers, tail...)

fmt.Println(numbers)
```

First of all, you need to create another slice, `tail`, to store the *tail* part of the original slice. You can't simply slice it and store it into another variable (this is called *shallow copy*), because slices are not arrays: they are a pointer to a part of the array and its length. If you slice `numbers` and store it in `tail`, when you change `numbers`, `tail` will also change, and that is not what you want. Instead, you want to create a new slice by appending it to an empty slice of ints.

This will be `tail` after the first append:

```
[26 53 58]
```

Now that you have put the tail aside, you append the head of `numbers` to the unpacked `inserted`. At this stage, `numbers` becomes this, because you take the first three elements and append `inserted` behind it:

```
[3 14 159 1000 2000 3000 4000]
```

Finally, you append `numbers` (which now consists of the head of the original slice and `inserted`) and the `tail`. This is what you should get:

```
[3 14 159 1000 2000 3000 4000 26 53 58]
```

Removing

Removing elements from a slice is very easy. If it's at the start or end of the slice, you simply reslice it accordingly to remove either the start or the end of the slice.

To take out the first element of the slice:

```
numbers := []int{3, 14, 159, 26, 53, 58}
numbers = numbers[1:] // remove element 0
fmt.Println(numbers)
```

When you run this, you will get:

```
[14 159 26 53 58]
```

Now take out the last element of the slice:

```
numbers := []int{3, 14, 159, 26, 53, 58}
numbers = numbers[:len(numbers)-1] // remove last element
fmt.Println(numbers)
```

When you run this code, you will get:

```
[3 14 159 26 53]
```

Removing elements in between two adjacent elements within a slice is quite straightforward too. You simply append the head of the original slice with the tail of the original slice, removing whatever is in between. In this case, you want to remove the element at index 2, which is 159:

```
numbers := []int{3, 14, 159, 26, 53, 58}
numbers = append(numbers[:2], numbers[3:]...)
fmt.Println(numbers)
```

When you run the code, you get this:

```
[3 14 26 53 58]
```

13.4 Making Arrays and Slices Safe for Concurrent Use

Problem

You want to make arrays and slices safe for concurrent use by multiple goroutines.

Solution

Use a mutex from the `sync` library to safeguard the array or slice. Lock the array or slice before modifying it, and unlock it after modifications are made.

Discussion

Arrays and slices are not safe for concurrent use. If you are going to share a slice or array between goroutines, you need to make it safe from race conditions. Go provides a `sync` package that can be used for this, in particular, `Mutex`.

Race conditions occur when a shared resource is used by multiple goroutines trying to access it at the same time:

```
var shared []int = []int{1, 2, 3, 4, 5, 6}

// increase each element by 1
func increase(num int) {
        fmt.Printf("[+%d a] : %v\n", num, shared)
        for i := 0; i < len(shared); i++ {
                time.Sleep(20 * time.Microsecond)
                shared[i] = shared[i] + 1
        }
        fmt.Printf("[+%d b] : %v\n", num, shared)
```

```
}

// decrease each element by 1
func decrease(num int) {
        fmt.Printf("[-%d a] : %v\n", num, shared)
        for i := 0; i < len(shared); i++ {
                time.Sleep(10 * time.Microsecond)
                shared[i] = shared[i] - 1
        }
        fmt.Printf("[-%d b] : %v\n", num, shared)
}
```

In this example, you have a slice of integers named `shared` that is used by two functions named `increase` and `decrease`. These two functions simply take each element in the shared slice and increase or decrease it by 1, respectively. However, before you increase or decrease the element, you wait for a very short period, with the `increase` function waiting for a longer time. This simulates the differences in timing between multiple goroutines. You print out the `shared` slice before you start modifying the shared element and also after you modify it to show the state of the shared slice before and after.

You call the `increase` and decrease functions from `main`, and you make each call to the functions a separate goroutine. At the end of the program, you wait a bit to let all the goroutines finish (else all goroutines will end when the program ends):

```
func main() {
        for i := 0; i < 5; i++ {
                go increase(i)
        }
        for i := 0; i < 5; i++ {
                go decrease(i)
        }
        time.Sleep(2 * time.Second)
}
```

When you run the program, you will see something like this:

```
[-4 a] : [1 2 3 4 5 6]
[-1 a] : [0 2 3 4 5 6]
[-2 a] : [0 1 3 4 5 6]
[-3 a] : [0 1 2 4 5 6]
[+0 a] : [-2 1 2 3 5 6]
[+1 a] : [-3 -1 2 3 4 6]
[-4 b] : [-2 -2 1 3 4 5]
[+3 a] : [-2 -2 0 3 4 5]
[+4 a] : [-1 -1 -1 1 4 5]
[-1 b] : [1 0 0 0 1 4]
[-2 b] : [1 0 0 0 1 3]
[-3 b] : [1 0 0 0 1 2]
[+2 a] : [1 0 0 0 1 2]
[-0 a] : [2 2 1 1 1 2]
```

```
[+0 b] : [1 2 3 2 1 3]
[-0 b] : [1 2 3 3 2 2]
[+1 b] : [1 2 3 4 4 3]
[+3 b] : [1 2 3 4 4 4]
[+4 b] : [1 2 3 4 4 5]
[+2 b] : [1 2 3 4 5 6]
```

A quick explanation of the output: The header in each line starts with – or +, depending on whether the `decrease` or `increase` function is called. The number after that is the sequence of the function call, and the letter a or b simply indicates the state of the shared slice at the start of the function or at the end of it.

Let's take the first line `[-4 a] : [1 2 3 4 5 6]`. This means the decrease function is called, and this is sequence 4, and the state of the shared slice is [1 2 3 4 5 6].

If you run it multiple times the result will be a bit different each time. You will notice that even though you spin out a goroutine in sequence (sending in the sequence number to `modify` each time), the sequence that executes is random, which is expected behavior. What you don't want is for the goroutines to overlap each other and for the shared slice to be incremented or decremented depending on which goroutine accesses it first.

Subsequently, after the loop the line that is printed is `[-4 b] : [-2 -2 1 3 4 5]` and you can see the first three elements of the shared slice are not what is expected! In case this is not clear, the slice should be [0 1 2 3 4 5] instead of [–2 –2 1 3 4 5].

Also, you will realize that the overlap even happens within the loop for increasing or decreasing the element.

How can you prevent such race conditions? Go has the `sync` package in the standard library that provides you with a *mutex*, or a mutual exclusion lock:

```
var shared []int = []int{1, 2, 3, 4, 5, 6}
var mutex sync.Mutex

// increase each element by 1
func increaseWithMutex(num int) {
        mutex.Lock()
        fmt.Printf("[+%d a] : %v\n", num, shared)
        for i := 0; i < len(shared); i++ {
                time.Sleep(20 * time.Microsecond)
                shared[i] = shared[i] + 1
        }
        fmt.Printf("[+%d b] : %v\n", num, shared)
        mutex.Unlock()
}

// decrease each element by 1
func decreaseWithMutex(num int) {
        mutex.Lock()
```

```
            fmt.Printf("[-%d a] : %v\n", num, shared)
            for i := 0; i < len(shared); i++ {
                    time.Sleep(10 * time.Microsecond)
                    shared[i] = shared[i] - 1
            }
            fmt.Printf("[-%d b] : %v\n", num, shared)
            mutex.Unlock()
    }
    }
```

Using it is quite simple. First you need to declare a mutex. Then, you call `Lock` on the mutex before you start modifying the shared slice. This will lock the shared slice such that nothing else can use it. When you're done, you call `Unlock` to unlock the mutex.

Here's the output if you call these functions from `main` as before:

```
[-4 a] : [1 2 3 4 5 6]
[-4 b] : [0 1 2 3 4 5]
[+0 a] : [0 1 2 3 4 5]
[+0 b] : [1 2 3 4 5 6]
[+1 a] : [1 2 3 4 5 6]
[+1 b] : [2 3 4 5 6 7]
[+2 a] : [2 3 4 5 6 7]
[+2 b] : [3 4 5 6 7 8]
[+3 a] : [3 4 5 6 7 8]
[+3 b] : [4 5 6 7 8 9]
[+4 a] : [4 5 6 7 8 9]
[+4 b] : [5 6 7 8 9 10]
[-0 a] : [5 6 7 8 9 10]
[-0 b] : [4 5 6 7 8 9]
[-1 a] : [4 5 6 7 8 9]
[-1 b] : [3 4 5 6 7 8]
[-2 a] : [3 4 5 6 7 8]
[-2 b] : [2 3 4 5 6 7]
[-3 a] : [2 3 4 5 6 7]
[-3 b] : [1 2 3 4 5 6]
```

The results are a lot more organized. The goroutines no longer overlap, and the increase and decrease of elements are orderly and consistent.

13.5 Sorting Arrays of Slices

Problem

You want to sort elements in an array or slice.

Solution

For `int`, `float64`, and `string` arrays or slices you can use `sort.Ints`, `sort.Float64s`, and `sort.Strings`. You can also use a custom comparator by using `sort.Slice`.

For structs, you can create a sortable interface by implementing the `sort.Interface` interface and then using `sort.Sort` to sort the array or slice.

Discussion

Arrays and slices are ordered sequences of elements. However, this doesn't mean they are sorted in any way; it only means the elements are always laid out in the same sequence. To sort the arrays or slices, you can use the various functions in the `sort` package.

For `int`, `float64`, and `string` you can use the corresponding `sort.Ints`, `sort.Float64s`, and `sort.Strings` functions:

```
integers := []int{3, 14, 159, 26, 53}
floats := []float64{3.14, 1.41, 1.73, 2.72, 4.53}
strings := []string{"the", "quick", "brown", "fox", "jumped"}

sort.Ints(integers)
sort.Float64s(floats)
sort.Strings(strings)

fmt.Println(integers)
fmt.Println(floats)
fmt.Println(strings)
```

If you run the code, this is what you will see:

```
[3 14 26 53 159]
[1.41 1.73 2.72 3.14 4.53]
[brown fox jumped quick the]
```

This is sorted in ascending order. What if you want to sort it in descending order? There is no ready-made function to sort in descending order, but you can easily use a simple `for` loop to reverse the sorted slice:

```
for i := len(integers)/2 - 1; i >= 0; i-- {
    opp := len(integers) - 1 - i
    integers[i], integers[opp] = integers[opp], integers[i]
}

fmt.Println(integers)
```

Simply find the middle of the slice, and then using a loop, exchange the elements with their opposite side, starting from that middle. If you run the preceding snippet, this is what you will get:

```
[159 53 26 14 3]
```

You can also use the `sort.Slice` function, passing in your `less` function:

```
sort.Slice(floats, func(i, j int) bool {
    return floats[i] > floats[j]
})
fmt.Println(floats)
```

This will produce the following output:

```
[4.53 3.14 2.72 1.73 1.41]
```

The `less` function, the second parameter in the `sort.Slice` function, takes in two parameters `i` and `j`, indices of the consecutive elements of the slice. It's supposed to return `true` if the element at `i` is less than the element at `j` when sorting.

What if the elements are the same? Using `sort.Slice` means the original order of the elements might be reversed (or remain the same). If you want the order to be consistently the same as the original, you can use `sort.SliceStable`.

The `sort.Slice` function works with slices of any type, so this means you can also sort custom structs:

```
people := []Person{
        {"Alice", 22},
        {"Bob", 18},
        {"Charlie", 23},
        {"Dave", 27},
        {"Eve", 31},
}
sort.Slice(people, func(i, j int) bool {
        return people[i].Age < people[j].Age
})
fmt.Println(people)
```

If you run the code you will get the following output, with the `people` slice sorted according to the ages of the people:

```
[{Bob 18} {Alice 22} {Charlie 23} {Dave 27} {Eve 31}]
```

Another way of sorting structs is by implementing the `sort.Interface`. Here's how you can do this for the `Person` struct:

```
type Person struct {
        Name string
        Age  int
}

type ByAge []Person

func (a ByAge) Len() int           { return len(a) }
func (a ByAge) Less(i, j int) bool { return a[i].Age < a[j].Age }
func (a ByAge) Swap(i, j int)      { a[i], a[j] = a[j], a[i] }
```

You want to sort a slice of structs, so you need to associate the interface functions to the slice, not the struct. Create a type named `ByAge` that is a slice of `Person` structs.

Next, you associate the `Len`, `Less`, and `Swap` functions to `ByAge`, making it a struct that implements `sort.Interface`. The `Less` method here is the same as the one used in the `sort.Slice` function earlier.

Using this is quite simple. You cast `people` to `ByAge`, and pass that into `sort.Sort`:

```
people := []Person{
        {"Alice", 22},
        {"Bob", 18},
        {"Charlie", 23},
        {"Dave", 27},
        {"Eve", 31},
}

sort.Sort(ByAge(people))
fmt.Println(people)
```

If you run this code, you will see the following results:

```
[{Bob 18} {Alice 22} {Charlie 23} {Dave 27} {Eve 31}]
```

Implementing `sort.Interface` is a bit long-winded, but there are certainly some advantages. For one, you can use `sort.Reverse` to sort by descending order:

```
sort.Sort(sort.Reverse(ByAge(people)))
fmt.Println(people)
```

This produces the following output:

```
[{Eve 31} {Dave 27} {Charlie 23} {Alice 22} {Bob 18}]
```

You can also use the `sort.IsSorted` function to check if the slice is already sorted:

```
sort.IsSorted(ByAge(people)) // true if it's sorted
```

The biggest advantage, though, is that using `sort.Interface` is a lot more performant than using `sort.Slice`, as shown by this simple benchmark:

```
func BenchmarkSortSlice(b *testing.B) {
        f := func(i, j int) bool {
                return people[i].Age < people[j].Age
        }
        b.ResetTimer()
        for i := 0; i < b.N; i++ {
                sort.Slice(people, f)
        }
}

func BenchmarkSortInterface(b *testing.B) {
        for i := 0; i < b.N; i++ {
                sort.Sort(ByAge(people))
        }
}
```

Here are the results of the benchmark:

```
$ go test -bench=BenchmarkSort
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch14_data_structures
BenchmarkSortSlice-10            9376766               108.9 ns/op
BenchmarkSortInterface-10       26790697                44.33 ns/op
PASS
ok      github.com/sausheong/gocookbook/ch14_data_structures    2.901s
```

As you can see, using `sort.Interface` is more efficient. This is because `sort.Slice` uses `any` as the first parameter. This means it takes in any structs but is less efficient.

13.6 Creating Maps

Problem

You want to create new maps.

Solution

Use the `map` keyword to declare it, and then use the `make` function to initialize it. Maps must be initialized before use.

Discussion

To create a map, you can use the `map` keyword:

```
var people map[string]int
```

This snippet declares a map named `people` that maps a key of type `string` to a value of type `int`. The `people` map can't be used yet since its zero-value is nil. To use it you need to initialize it with the `make` method:

```
people = make(map[string]int)
```

If it looks silly that you have to repeat `map[string]int` in both the declaration and initialization, you should do both at the same time:

```
people := make(map[string]int)
```

This will create an empty map. To populate the map you can map a string to an int:

```
people["Alice"] = 22
```

You can also initialize the map this way:

```
people := map[string]int{
        "Alice":   22,
        "Bob":     18,
```

```
        "Charlie": 23,
        "Dave":    27,
        "Eve":     31,
}
```

If you print the map out, this is how it will look:

```
map[Alice:22 Bob:18 Charlie:23 Dave:27 Eve:31]
```

13.7 Accessing Maps

Problem

You want to access keys and values in a map.

Solution

Use the key within square brackets to access the value in a map. You can also use a `for ... range` loop to iterate through the map.

Discussion

Accessing the values given a key is straightforward. Just use the key within square brackets to access the values:

```
people := map[string]int{
        "Alice":   22,
        "Bob":     18,
        "Charlie": 23,
        "Dave":    27,
        "Eve":     31,
}

people["Alice"] // 22
```

What if the key doesn't exist? Nothing happens: Go simply returns the zero-value of the value type. In this case the zero-value of an integer is 0, so if you do this:

```
people["Nemo"] // 0
```

it will simply return a 0. This might not be what you're looking for (especially if 0 is a valid response) so there is a mechanism to check if the key exists or not:

```
age, ok := people["Nemo"]
if ok {
        // do whatever you want if the value exists
}
```

The *comma, ok* pattern is commonly used in Go and can be used here to check if the key exists in the map. If the key exists, `ok` becomes true, else `ok` is false. The `ok` variable is not a keyword. You can use any variable name, because it's using the

multiple value assignment. The value is still returned but since you know the key doesn't exist and it's just a zero-value, you probably would not use it.

You can also use a `for ... range` loop to iterate through a map, just like you did with arrays and slices, except instead of getting the index and the element, you get the key and the value:

```
for k, v := range people {
        fmt.Println(k, v)
}
```

Running this code will give you the following output:

```
Alice 22
Bob 18
Charlie 23
Dave 27
Eve 31
```

If you want just the keys, you can leave out the second value you get from the range:

```
for k := range people {
        fmt.Println(k)
}
```

You will get this output:

```
Alice
Bob
Charlie
Dave
Eve
```

What if you want just the values? There is no special way of getting just the values; you have to use the same mechanism and put them all in a slice:

```
var values []int
for _, v := range people {
        values = append(values, v)
}
fmt.Println(values)
```

You will get this output:

```
[22 18 23 27 31]
```

13.8 Modifying Maps

Problem

You want to modify or remove elements in a map.

Solution

Use the `delete` function to remove key-value pairs from a map. To modify the value, just reassign the value.

Discussion

Modifying a value is simply overriding the existing value:

```
people["Alice"] = 23
```

The value of `people["Alice"]` will become 23.

To remove a key, Go provides a built-in function named `delete`:

```
delete(people, "Alice")
fmt.Println(people)
```

This will be the output:

```
map[Bob:18 Charlie:23 Dave:27 Eve:31]
```

What happens if you try to delete a key that doesn't exist? Nothing happens.

13.9 Sorting Maps

Problem

You want to sort a map by its keys.

Solution

Get the keys of the map in a slice and sort that slice. Then, using the sorted slice of keys, iterate through the map again.

Discussion

Maps are unordered. This means each time you iterate through a map, the order of the key-value pairs might not be the same as the previous time. So how can you ensure that it's the same each time?

First, extract the keys into a slice:

```
var keys []string
for k := range people {
        keys = append(keys, k)
}
```

Then sort the keys accordingly. In this case, you want to sort by descending order:

```
// sort keys by descending order
for i := len(keys)/2 - 1; i >= 0; i-- {
        opp := len(keys) - 1 - i
        keys[i], keys[opp] = keys[opp], keys[i]
}
```

Finally, you can access the map by the descending order of the keys:

```
for _, key := range keys {
        fmt.Println(key, people[key])
}
```

When you run the code, you will see this:

```
Eve 31
Dave 27
Charlie 23
Bob 18
Alice 22
```

CHAPTER 14

More Data Structure Recipes

14.0 Introduction

Chapter 12 discussed structs and Chapter 13 discussed arrays, slices, and maps. These are the four basic data structures in Go. Unlike many other programming languages, Go does not provide other basic data structures (although there is the `container` package in the standard library it has very few implementations). In your daily programming tasks, you often will have to use some other data structures that are not provided either in the language or in the standard library, and it's not very difficult to re-create them.

In this chapter, you will be creating a few common data structures with Go:

- Queue
- Stack
- Set
- Linked list
- Heap
- Graph

Each recipe will start by explaining what that data structure is, then go through how to build one from the ground up.

None of the data structures are concurrency-safe. This is because fundamentally they are built on the three basic Go data structures—arrays, slices, and maps—and these are not concurrency-safe. To avoid race conditions, you can add a *mutex* to the data structure and use it to lock the data structure before any reads or writes. The `sync` package has a `RWMutex` that you can use for this purpose.

A word on the terminology: these recipes use the term *list* to refer to a linear, ordered sequence of items. Items within a list are called *elements*. Arrays and slices in Go are lists. Similarly, a *graph* refers to a group of items that are connected. Items within a graph are called *nodes*, and the connections between nodes are called *edges*.

14.1 Creating Queues

Problem

You want to create a queue data structure.

Solution

Wrap a struct around a slice. Create queue functions on the struct.

Discussion

A queue is a first-in-first-out (FIFO) ordered list. You add elements at the back of the queue and get elements at the front of the queue. You can visualize a queue to be exactly like what its name is—a line of shoppers queueing at the counter at the supermarket, waiting to pay for their purchases. When a new shopper joins the queue, they join the back of the queue. When a shopper completes the purchase, they exit the queue from the front. Figure 14-1 shows a queue.

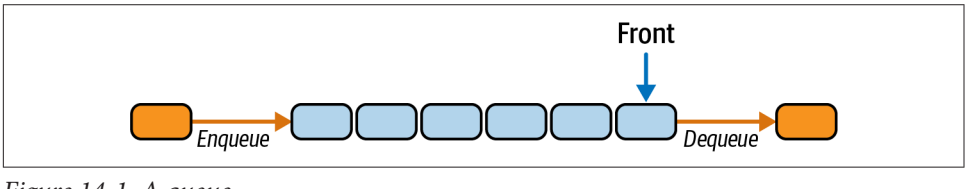

Figure 14-1. A queue

Queues are one of the simplest data structures. They are commonly used as a buffer or queueing system.

Here are the functions associated with a queue:

Enqueue
: Add an element to the back of the queue.

Dequeue
: Remove an element at the front of the queue.

Peek
: Get the element at the front of the queue without removing it from the queue.

Size
: Get the size of the queue.

IsEmpty
: Check if the queue is empty.

You can easily implement a queue using a slice:

```
type Queue struct {
        elements []any
}
```

In this code you are using `any` elements so it can be any type.

Enqueuing an element is as simple as appending the element to the back of the slice. This means you consider the front of the slice to be the front of the queue as well:

```
func (q *Queue) Enqueue(el any) {
        q.elements = append(q.elements, el)
}
```

To dequeue an element, take the element at index 0 and remove it from the queue by reslicing. Of course, if the queue is empty, you should return an error:

```
func (q *Queue) Dequeue() (el any, err error) {
        if q.IsEmpty() {
                err = errors.New("empty queue")
                return
        }
        el = q.elements[0]
        q.elements = q.elements[1:]
        return
}
```

Peeking at the queue returns the element at index 0:

```
func (q *Queue) Peek() (el any, err error) {
        if q.IsEmpty() {
                err = errors.New("empty queue")
                return
        }
        el = q.elements[0]
        return
}
```

Finally, checking the size and if the queue is empty is quite straightforward:

```
func (q *Queue) IsEmpty() bool {
        return q.Size() == 0
}

func (q *Queue) Size() int {
        return len(q.elements)
}
```

14.2 Creating Stacks

Problem

You want to create a stack data structure.

Solution

Wrap a struct around a slice. Create stack functions on the struct.

Discussion

A stack is a last-in-first-out (LIFO) ordered list. You add elements at the top of the stack and get elements from the top of the stack as well. A good way to visualize a stack is how t-shirts are folded and placed on top of each other on a shelf or table in a shop. When you want to add a t-shirt, you add it to the top of the stack of t-shirts. If you want to take one, you take it from the top of the stack as well.

Stacks are very important in programming, especially in memory management. They're also used for creating recursive functions, expression, evaluation, and backtracking.

Here are the functions associated with a stack:

Push
: Add an element to the top of the stack.

Pop
: Remove an element at the top of the stack.

Peek
: Get the element at the top of the stack without removing it.

Size
: Get the size of the stack.

IsEmpty
: Check if the stack is empty.

Figure 14-2 shows a stack.

There are a few ways to implement a stack but here you'll simply be using a single slice:

```
type Stack struct {
        elements []any
}
```

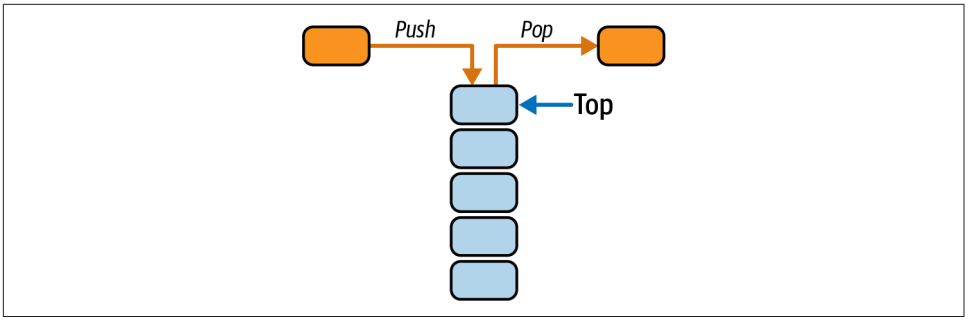

Figure 14-2. A stack

In the preceding code, you're using `any` as elements in the stack:

```
func (s *Stack) Push(el any) {
        s.elements = append(s.elements, el)
}
```

Pushing a new element to the top of the stack is the same as with the queue in Recipe 14.1, by appending the new element to the slice. By doing so, you're assuming the last element in the slice to be the top of the stack.

This means you will also pop from the same place:

```
func (s *Stack) Pop() (el any, err error) {
        if s.IsEmpty() {
                err = errors.New("empty stack")
                return
        }
        el = s.elements[len(s.elements)-1]
        s.elements = s.elements[:len(s.elements)-1]
        return
}
```

To pop the stack, take the last element of the slice, then reslice it to exclude that last element. Of course, you can't pop the stack if it's empty, so you need to return an error when that happens:

```
func (s *Stack) Peek() (el any, err error) {
        if s.IsEmpty() {
                err = errors.New("empty queue")
                return
        }
        el = s.elements[len(s.elements)-1]
        return
}
```

Peeking the stack is nothing more than returning the last element of the slice without reslicing:

```
func (s *Stack) IsEmpty() bool {
        return s.Size() == 0
}

func (s *Stack) Size() int {
        return len(s.elements)
}
```

Finally, getting the size of the stack and checking if it's empty are the same as with the queue.

14.3 Creating Sets

Problem

You want to create a set data structure.

Solution

Wrap a struct around a map. Create set functions on the struct.

Discussion

A set is an unordered data structure that has only unique elements. It implements the mathematical concept of a finite set and the operations around it. The basic functions associated with sets include:

Add
: Add a new element to the set.

Remove
: Remove an existing element from the set.

IsEmpty
: Check if the set is empty.

Size
: Get the size of the set.

In addition to these basic functions are the set operations, which are the implementations of mathematical operations:

Has
: Check if an element is a member of a given set.

Union
: Given two or more sets, the union of the sets is the set that consists of elements that are in any of those sets.

Intersection
: Given two or more sets, the intersection of sets is the set that consists of elements that are in all the sets.

Difference
: Given two sets A and B, the difference A – B consists of elements that are in set A but not set B.

IsSubset
: Given two sets A and B, check if every element in set B is in set A.

Start with the implementation of the `Set` struct:

```
type Set struct {
        elements map[any]bool
}
```

Map keys are unordered and are also unique so it's no surprise that you implement `Set` with a map. In this implementation, the key is the element in the set, while you don't care about the value at all.

However, because you're using a map to represent a set, you need to use a separate function to create a new set. This is because maps need to be initialized before they can be used:

```
func NewSet() Set {
        return Set{elements: make(map[any]bool)}
}
```

In the `NewSet` function, you create a `Set` by using `make` to initialize the internal map.

Add

Adding an element to the set is simply adding an element to the internal map. You don't care what is used for the value. In the following code, simply set it to false:

```
func (s *Set) Add(el any) {
        s.elements[el] = false
}
```

Remove

Similarly, use `delete` to remove the element from the internal map:

```
func (s *Set) Remove(el any) {
        delete(s.elements, el)
}
```

IsEmpty and Size

To check if the set is empty and also to get the size of the set, use `len` on the internal map:

```
func (s *Set) IsEmpty() bool {
        return s.Size() == 0
}

func (s *Set) Size() int {
        return len(s.elements)
}
```

List

Also, as a convenience, you want to convert the set into a list, which is just a slice representation of the keys in the internal map:

```
func (s *Set) List() (list []any) {
        for k := range s.elements {
                list = append(list, k)
        }
        return
}
```

Has

Next are the set operations. The first set operation is the membership operation, where you want to check if a set has an element. You implement this with the `Has` method. This is an important method because you'll be using the membership operation in the other set operations:

```
func (s Set) Has(el any) (ok bool) {
        _, ok = s.elements[el]
        return
}
```

To implement the `Has` method, you use the comma, ok notation on the internal map, which tells you whether or not the key exists in the map. You ignore the value altogether.

Union

Next is the union of sets. The `Union` function takes in a variable number of sets. Union is an *OR* relationship on all the sets.

Take the first set and set it as the seed set for the union of all the sets. Remember, the union has all the elements in all the sets, so it must also include all the elements in the seed set. However, you don't want to modify the first set directly—you want to make

a copy of it instead of modifying the first set. To do that you create a `Copy` function, which allows you to make a copy of a set:

```
func (s *Set) Copy() (u Set) {
        u = NewSet()
        for k := range s.elements {
                u.Add(k)
        }
        return
}
```

With that, you iterate through the rest of the sets and add the elements in each of the sets to the seed set. This will result in a final set that has all the elements from all the sets:

```
func Union(sets ...Set) (u Set) {
        u = sets[0].Copy()
        for _, set := range sets[1:len(sets)] {
                for k := range set.elements {
                        u.Add(k)
                }
        }
        return
}
```

Intersect

The intersection of sets gives you a set in which elements exist in every given set. The `Intersect` function takes in a variable number of sets and returns the intersection of all the sets. Intersection is an *AND* relationship on all the sets:

```
func Intersect(sets ...Set) (i Set) {
        i = sets[0].Copy()
        for k := range i.elements {
                for _, set := range sets[1:len(sets)] {
                        if !set.Has(k) {
                                i.Remove(k)
                        }
                }
        }
        return
}
```

As before, you take the first set as the seed set for the intersection of all the sets. The seed set must have all the elements in the intersection (because an element exists in the intersection only if it exists in all the sets). You just need to remove those that are not in the intersection from the seed set. As before, you don't want to modify the first set but make a copy of it instead.

Iterate through every element in the seed set and check if it exists in each of the rest of the sets. If a set doesn't have the element, remove the element from the seed.

The `Difference` method on a set subtracts the given set from itself and returns the difference:

```
func (s Set) Difference(t Set) Set {
        for k := range t.elements {
                if s.Has(k) {
                        delete(s.elements, k)
                }
        }
        return s
}
```

Given a set `s` and you want to subtract the set `t` from it, you iterate through each element in `t` and check if it is found in `s`. If it is, you remove it from `s`. Once you're done with all the elements in `t`, you return `s`.

Finally, you also want to check if set `s` is a subset of `t`:

```
func (s Set) IsSubset(t Set) bool {
        for k := range s.elements {
                if !t.Has(k) {
                        return false
                }
        }
        return true
}
```

To do this, iterate through each element in set `s` and check if it is also in set `t`. If there is an element in set `s` that is not found in set `t`, this means `s` is not a subset of `t`.

14.4 Creating Linked Lists

Problem

You want to create a linked list data structure.

Solution

Create an element struct that has a pointer to the next element. Wrap another struct around the first element to create a linked list.

Discussion

A linked list is a linear collection of elements that has each element pointing to the next element. This is different from a list because in lists the elements are next to each other (incrementing the index of the current element will give you access to the next element) while for a linked list this is not necessarily so. As a result, inserting or removing an element is faster because you don't need to restructure the data

structure, unlike in a list. On the other hand, accessing random elements in a linked list is slower because elements are not indexed, so you need to iterate through the elements until the correct one is found. Figure 14-3 shows a linked list.

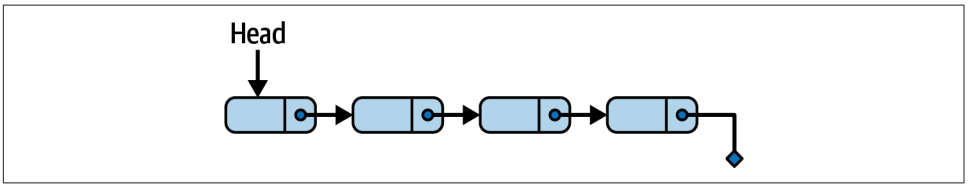

Figure 14-3. A linked list

This recipe will go through a simple implementation of a singly linked list. The payload for the linked list is just a string.

The functions for the singly linked list are:

Add
: Add a new element to the end of the linked list.

Insert
: Insert a new element into a specific position within the linked list.

Delete
: Remove a specific element from the linked list.

Find
: Find and return an element from the linked list.

In previous recipes in this chapter, you used only one struct to represent the data structure. The elements in those data structures are of type `any`. In a linked list, you will use two structs—one to represent the data structure, `LinkedList`, and another to represent an element in the data structure, `Element`:

```
import "golang.org/x/exp/constraints"

type Element[T constraints.Ordered] struct {
        value T
        next  *Element[T]
}

type LinkedList[T constraints.Ordered] struct {
        head *Element[T]
        size int
}
```

You need to do this because each element in the linked list points to the next; therefore it needs to keep a pointer to the next. You create an `Element` struct to represent an element, with a `next` field pointing to the next element. The `Element`

struct also has a value, which is of type `T`, constrained by the `constraints.Ordered` type. This means you can use the ordering operators (<, >, <=, >=) on `Element` values.

You create a `LinkedList` struct to keep track of the head of the linked list and its size. You need this struct to associate all the linked list functions.

Add

Adding to a linked list means adding a new element to the head of the linked list and moving the current head down the chain:

```
func (l *LinkedList[T]) Add(el *Element[T]) {
        if l.head == nil {
                l.head = el
        } else {
                el.next = l.head
                l.head = el
        }
        l.size++
}
```

If it's an empty linked list with the head pointing to nil, set the new element to the head. If not, set the next element of the new element to the head, and set the head to the new element.

Insert

Inserting a new element requires you to know where you want to insert it. The `Insert` method has two parameters—the new element and a marker that indicates that the new element should be inserted after it:

```
func (l *LinkedList[T]) Insert(el *Element[T], marker T) error {
        for current := l.head; current.next != nil; current = current.next {
                if current.value == marker {
                        el.next = current.next
                        current.next = el
                        l.size++
                        return nil
                }
        }
        return errors.New("element not found")
}
```

Figure 14-4 shows how to insert a new element between two elements of a linked list.

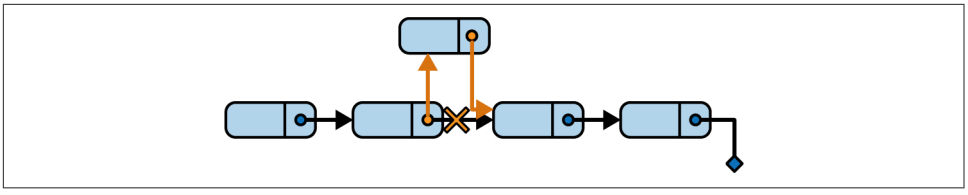

Figure 14-4. Inserting into a linked list

Iterate the linked list until you find the marker. Then set the new element's next element to the current element's next element, then set the current element's next element to the new element. This effectively inserts the new element between the current element and its next element.

Delete

Deleting an element is a bit trickier. As before you need to iterate through the linked list to find the element to delete. However, unlike previously you need to keep track of the previous element as well as the current element:

```
func (l *LinkedList[T]) Delete(el *Element[T]) error {
        prev := l.head
        current := l.head
        for current != nil {
                if current.value == el.value {
                        if current == l.head {
                                l.head = current.next
                        } else {
                                prev.next = current.next
                        }
                        l.size--
                        return nil
                }
                prev = current
                current = current.next
        }
        return errors.New("element not found")
}
```

Figure 14-5 shows how an element can be deleted from a linked list.

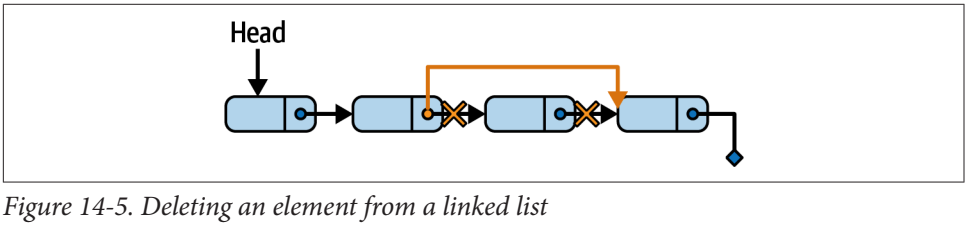

Figure 14-5. Deleting an element from a linked list

Once you reach the element you want to delete, set the previous element next to the current element's next. This effectively bypasses the current element and therefore removes it from the linked list. Of course, if the element to delete is the head of the linked list, you simply make the next element the head instead.

Find

As before, finding an element in the linked list means you need to iterate through the linked list and check each element:

```
func (l *LinkedList[T]) Find(value T) (el *Element[T], err error) {
        for current := l.head; current.next != nil; current = current.next {
                if current.value == value {
                        el = current
                        break
                }
        }
        if el == nil {
                err = errors.New("element not found")
        }
        return
}
```

Once the element is found, it will be returned.

List

A useful method to have is one that converts the linked list into a slice. This allows you the best of both worlds: the elements in a linked list as well as in a slice. However, the linked list and slice are not synchronized with each other; they are simply different ways of organizing the elements:

```
func (l *LinkedList[T]) List() (list []*Element[T]) {
        if l.head == nil {
                return []*Element[T]{}
        }
        for current := l.head; current != nil; current = current.next {
                list = append(list, current)
        }
        return
}
```

If you append a new `Element` to the slice, that element doesn't get added to the linked list, and if you insert a new `Element` into the linked list, it's not going to be reflected in the slice, unless you call the `List` method again.

Finally, the `Size` and `IsEmpty` methods take directly from the `size` field in the linked list to determine the size of the linked list and if the linked list is empty, respectively:

```
func (l *LinkedList[T]) IsEmpty() bool {
        return l.size == 0
```

```
}

func (l *LinkedList[T]) Size() int {
        return l.size
}
```

The standard library has a `container` package with a few data structure implementations, and it includes a doubly linked list. If you're looking for a linked list, this would be a good place to start.

14.5 Creating Heaps

Problem

You want to create a min heap data structure.

Solution

Wrap a struct around a slice of elements to represent a heap. After each push or pop on the heap, rearrange the heap structure.

Discussion

A *heap* is a tree-like data structure that satisfies the heap property. The *heap property* is defined such that each node in a tree has a key that is greater (or less) than or equal to its parent.

There are two types of heaps. The first is the *min heap*, where the heap property is such that the key of the parent node is always smaller than that of the child nodes. The second is the *max heap*, where the heap property, as you would expect, is such that the key of the parent node is always larger than that of the child nodes.

The heap is commonly used as a priority queue. A *priority* queue, as the name suggests, is a queue where each element is given a priority. Priority queues can be implemented in other ways besides using heaps but heaps are so commonly used that sometimes priority queues are simply called heaps.

The heap functions are simple:

Push
: Add a node to the top of the heap.

Pop
: Take the node at the top of the heap.

A popular heap implementation is the *binary heap*, where each node has at most two child nodes. Let's see how a binary heap can be implemented. You might be

surprised that heaps are commonly implemented in the same way a queue or a stack is implemented, using a slice!

In the max heap implementation in this recipe, use an integer as the node for simplicity:

```
type Heap[T constraints.Ordered] struct {
        nodes []T
}
```

Figure 14-6 shows a max heap.

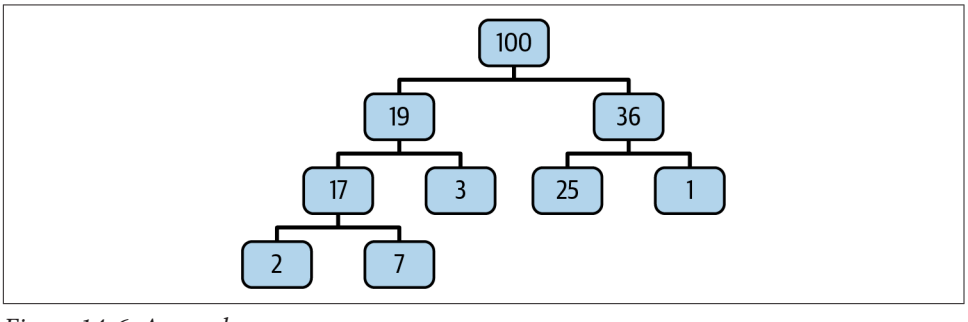

Figure 14-6. A max heap

Figure 14-7 shows how the max heap in Figure 14-6 would look when laid out in the form of a list.

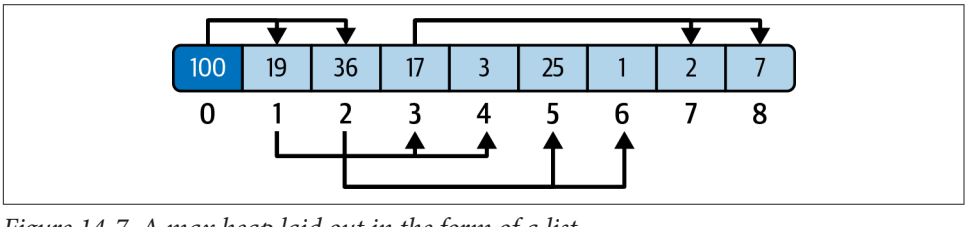

Figure 14-7. A max heap laid out in the form of a list

Take a closer look at the list. You can see that 100 is the parent of 19 and 36, 19 is the parent of 17 and 3, and so on. How you find the child node (if any) is basically to double the index of the parent, and add either 1 or 2 (left or right). In other words:

```
Left child = (2 * parent) + 1
Right child = (2 * parent) + 2
```

In reverse, to find the parent, subtract 1 from the child and divide by 2. Because it's a binary tree and it's an integer, it will return the nearest integer:

```
parent = (child - 1)/2
```

This is what the code looks like:

```
func parent(i int) int {
        return (i - 1) / 2
}

func leftChild(i int) int {
        return 2*i + 1
}

func rightChild(i int) int {
        return 2*i + 2
}
```

The two main functions of a heap are also very similar. Push adds a new node to the top of the heap, while pop removes the top of the heap. However, unlike a stack, the node that is returned by pop will always be the smallest or the largest in the heap, depending on whether it's a min heap or a max heap—that's how the heap works.

As a result, every time you push or pop an element on the heap, the heap needs to reorganize itself.

Here's the `Push` method:

```
func (h *Heap[T]) Push(ele T) {
        h.nodes = append(h.nodes, ele)
        i := len(h.nodes) - 1
        for ; h.nodes[i] > h.nodes[parent(i)]; i = parent(i) {
                h.swap(i, parent(i))
        }
}
```

The algorithm for push is straightforward. Assuming you're doing a max heap, this is how it works:

1. Append the new element to the list.
2. Take the last element of the list, and make it the current element.
3. Check if it is larger than the parent. If yes, swap it with the parent.
4. Make the parent the current element and loop until the parent is no longer larger than the current element.

This will result in the newly added node bubbling up the heap until it is at a position where it's smaller than the parent but larger than both the children.

If you're concerned about the sibling, don't be. If it's larger than the parent, it'll be larger than the sibling. If it's not, it doesn't matter. In a max heap, it doesn't matter which sibling is left or right as long as both are smaller than the parent. This means there are many possible ways a heap can organize itself.

Now here's the Pop method:

```
func (h *Heap[T]) Pop() (ele T) {
        ele = h.nodes[0]
        h.nodes[0] = h.nodes[len(h.nodes)-1]
        h.nodes = h.nodes[:len(h.nodes)-1]
        h.rearrange(0)
        return
}
```

To recap, pop means you take out the top node of the heap, and you need to reorganize the heap after that. There is a bit more effort for popping the top of the heap, which involves recursion, but it's quite straightforward too.

This is how it works for a max heap:

1. Take out the top element (this means removing the element at index 0 of the list).
2. Take the last element of the list and move that to the top of the heap.
3. Call the recursive function to rearrange the heap, passing it the index of the top of the heap (this will be 0).

This is the recursive function:

```
func (h *Heap[T]) rearrange(i int) {
        largest := i
        left, right, size := leftChild(i), rightChild(i), len(h.nodes)

        if left < size && h.nodes[left] > h.nodes[largest] {
                largest = left
        }
        if right < size && h.nodes[right] > h.nodes[largest] {
                largest = right
        }
        if largest != i {
                h.swap(i, largest)
                h.rearrange(largest)
        }
}

func (h *Heap[T]) swap(i, j int) {
        h.nodes[i], h.nodes[j] = h.nodes[j], h.nodes[i]
}
```

The recursive algorithm works this way:

1. You start, assuming the element at the given index will be the largest.
2. You compare the left and right children of this element with itself.

3. If either the left or right child is larger than itself, you make the left or right child the largest by swapping out the elements and calling the recursive function with the new largest element.

This bubbles the last node down to its natural position. As you're doing this, you are also forcing the children of the original top of the heap to compare to see which one will go to the top of the heap (it must be either one of them since they are the next largest).

As with other data structures, there are methods to tell the size of the heap and to check if it's empty or not:

```
func (h *Heap) Size() int {
        return len(h.nodes)
}

func (h *Heap) IsEmpty() bool {
        return h.Size() == 0
}
```

The standard library's `container` package includes a `heap` package that provides heap operations for any type that implements its interface. If you need a heap, you might also consider using this package.

14.6 Creating Graphs

Problem

You want to create a weighted graph data structure.

Solution

Create structs for nodes and edges and place them in a `Graph` struct. Create and attach functions to `Graph` to create nodes and edges for the graph.

Discussion

Graphs are very common nonlinear data structures that are used everywhere to map relationships between entities. A graph consists of nodes and edges where edges connect two nodes and often represent the relationship between two nodes. Nodes are often given a name and sometimes a value.

In this recipe, you'll implement a type of graph called the *undirected weighted graph*, where the edges are also associated with a value, and the edges connect the nodes both ways. As you probably realize, there are many other kinds of graphs, but you can use the same techniques to implement them. Figure 14-8 shows an undirected weighted graph.

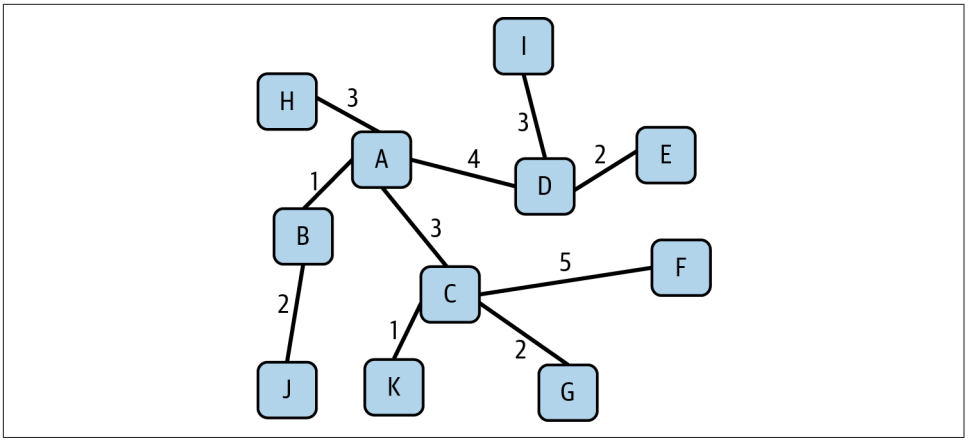

Figure 14-8. An undirected weighted graph

Here are the basic functions of a graph:

`AddNode`
: Add a new node into the graph

`AddEdge`
: Add a new edge that connects two nodes

`RemoveNode`
: Remove an existing node from the graph

`RemoveEdge`
: Remove an existing edge from the graph

You will use three structs to implement the weighted graph. The `Node` struct represents a node, with a given name. The `Edge` struct has a pointer to a node and also a weight. It might look odd but there is a reason for this:

```
type Node struct {
        name  string
}

type Edge struct {
        node   *Node
        weight int
}

type Graph struct {
        Nodes []*Node
        Edges map[string][]*Edge // key is node name
}
```

The last is the `Graph` struct, which represents the weighted graph. The `Graph` struct has a slice of pointers to `Node` structs. It also has a map that has a `string` key that associates with a slice of pointers to `Edge` structs. The key for this map is the name of the node and the value is a slice of `Edge` structs.

Edges are implemented using a map so you can't create a new `Graph` struct without also initializing the `Edges` field. To do this you have a function to create a new `Graph` struct:

```
func NewGraph() *Graph {
        return &Graph{
                Edges: make(map[string][]*Edge),
        }
}
```

AddNode

Adding a new node is very simple; just use the `append` function to add to the slice of existing nodes:

```
func (g *Graph) AddNode(n *Node) {
        g.Nodes = append(g.Nodes, n)
}
```

AddEdge

Adding a new edge is simple as well. Add a new key-value pair in the `Edges` field by creating a new `Edge` struct and appending it to the value:

```
func (g *Graph) AddEdge(n1, n2 *Node, weight int) {
        g.Edges[n1.name] = append(g.Edges[n1.name], &Edge{n2, weight})
        g.Edges[n2.name] = append(g.Edges[n2.name], &Edge{n1, weight})
}
```

Now that you can create a graph and add nodes and edges, you can build a simple graph like the one in Figure 14-9.

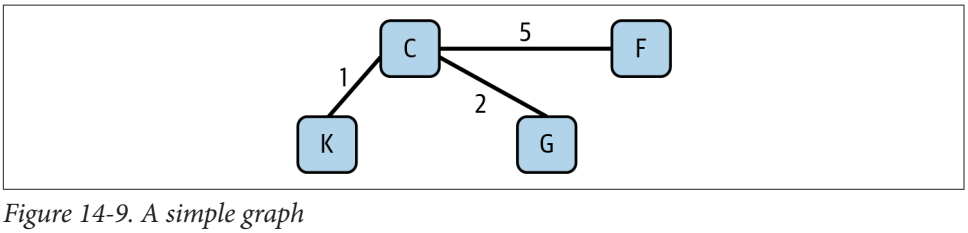

Figure 14-9. A simple graph

Here's the code:

```
graph := NewGraph()
c := &Node{"C"}
graph.AddNode(c)
```

```
k := &Node{"K"}
graph.AddNode(k)
g := &Node{"G"}
graph.AddNode(g)
f := &Node{"F"}
graph.AddNode(f)

graph.AddEdge(c, k, 1)
graph.AddEdge(c, g, 2)
graph.AddEdge(c, f, 5)
```

RemoveEdge

Removing an edge is a bit more involved. The edges are in a slice that are the values in the `Edges` map, so you need to iterate through them and mark it if it's to be removed. The `r` variable will store the index of the edge to be removed from the slice, else it will be –1:

```
func (g *Graph) RemoveEdge(n1, n2 string) {
        removeEdge(g, n1, n2)
        removeEdge(g, n2, n1)
}

func removeEdge(g *Graph, m, n string) {
        edges := g.Edges[m]
        r := -1
        for i, edge := range edges {
                if edge.node.name == n {
                        r = i
                }
        }
        if r > -1 {
                edges[r] = edges[len(edges)-1]
                g.Edges[m] = edges[:len(edges)-1]
        }
}
```

Once you find out where the edge is, take the last element in the slice and place it at the index, effectively removing that edge. After that, reslice to truncate the last element.

You have to do this twice to remove edges from both directions since this is an undirected graph.

RemoveNode

Removing nodes is also a bit more involved but very much the same as removing edges. First, you need to find out the index of the node to remove. Once you do that you take the last element in the slice and place it at the index, then truncate the last element. This effectively removes the node from the slice of nodes:

```
func (g *Graph) RemoveNode(name string) {
        r := -1
        for i, n := range g.Nodes {
                if n.name == name {
                        r = i
                }
        }
        if r > -1 {
                g.Nodes[r] = g.Nodes[len(g.Nodes)-1] // remove the node
                g.Nodes = g.Nodes[:len(g.Nodes)-1]
        }
        delete(g.Edges, name) // remove the edge from one side
        // remove the edge from the other side
        for n := range g.Edges {
                removeEdge(g, n, name)
        }
}
```

You also need to remove the edges that are connected to the node. First, delete the edge from the `Edges` map. Then go through the rest of the other key-value pairs and remove the node accordingly.

14.7 Finding the Shortest Path on a Graph

Problem

You want to find the shortest path between two nodes on a weighted graph.

Solution

Use Dijkstra's algorithm to find the shortest path between two nodes. Dijkstra's algorithm also uses a priority queue, which can be implemented using a min heap.

Discussion

The graph is one of the most commonly used data structures, often used to model many different problems. In social media applications like Facebook, how you and your friends are connected is often mapped to undirected graphs. In fact, this is often called the social graph, for a good reason. In mapping applications like Google Maps, the intersection of two or more roads are considered a node and the roads between the nodes are considered edges, where the map can be considered one big graph itself. One of the most popular algorithms, famously used by Google's search engine, is the *page rank algorithm*. This algorithm considers web pages to be nodes while the hyperlinks between the pages are edges.

As you probably realize, plenty of algorithms rely on graphs. One of the most well-known algorithms on graphs is *Dijkstra's algorithm*, sometimes called the *shortest*

path algorithm. As the name suggests, this simple yet effective algorithm finds the shortest path between two nodes in a graph.

Edsger Dijkstra was one of the giants in computer science, with many contributions to operating systems, including distributed computing, concurrent systems, structured programming, and graph algorithms. In 1959, Dijkstra published a three-page article titled "A Note on Two Problems in Connexion with Graphs," in *Numerische Mathematik*. In this article, he explained the algorithm to find the shortest path in a graph between any two given nodes. This eventually became what is known as Dijkstra's algorithm.

Say you want to travel from London to Istanbul and you figure out the following possible routes through the different major cities in Europe. Figure 14-10 shows a map of Europe, with the cities as nodes of a weighted directed graph, and flight routes between them as edges. The flight times between two cities are the edge weights.

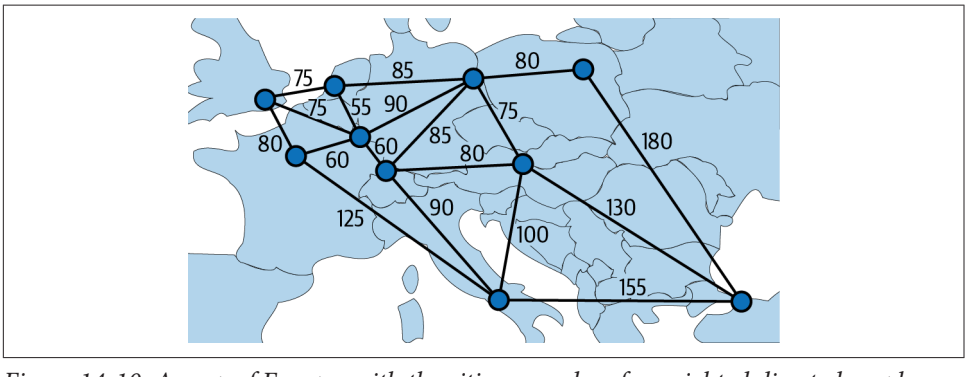

Figure 14-10. A map of Europe, with the cities as nodes of a weighted directed graph, and flight routes between them as edges

You have figured out the amount of time to fly from these cities to other cities but now you want to find the shortest amount of time needed to travel from London to Istanbul.

This is where Dijkstra's algorithm comes in handy, and it's probably the most popular algorithm to solve the shortest path problem. The algorithm itself is quite simple. Dijkstra famously came up with the algorithm without pen and paper, in about 20 minutes, while accompanying his fiancee shopping in Amsterdam.

You need two data structures for Dijkstra's algorithm—a weighted graph and a min heap (explained in Recipes 14.5 and 14.6, so please read these recipes if you haven't already). The only change you need is the `Node` struct in the weighted graph, to add a `through` pointer to a `Node`. This field, when not nil, points to the previous node in the shortest path:

```
type Node struct {
        name    string
        value   int
        through *Node
}
```

Before you start with the algorithm, you need to populate the graph accordingly:

```
func buildGraph() *Graph {
        graph := NewGraph()
        nodes := make(map[string]*Node)
        names := []string{"London", "Paris", "Amsterdam", "Luxembourg",
                "Zurich", "Rome", "Berlin", "Vienna", "Warsaw", "Istanbul"}
        for _, name := range names {
                n := &Node{name, math.MaxInt, nil}
                graph.AddNode(n)
                nodes[name] = n
        }
        graph.AddEdge(nodes["London"], nodes["Paris"], 80)
        graph.AddEdge(nodes["London"], nodes["Luxembourg"], 75)
        graph.AddEdge(nodes["London"], nodes["Amsterdam"], 75)
        graph.AddEdge(nodes["Paris"], nodes["Luxembourg"], 60)
        graph.AddEdge(nodes["Paris"], nodes["Rome"], 125)
        graph.AddEdge(nodes["Luxembourg"], nodes["Berlin"], 90)
        graph.AddEdge(nodes["Luxembourg"], nodes["Zurich"], 60)
        graph.AddEdge(nodes["Luxembourg"], nodes["Amsterdam"], 55)
        graph.AddEdge(nodes["Zurich"], nodes["Vienna"], 80)
        graph.AddEdge(nodes["Zurich"], nodes["Rome"], 90)
        graph.AddEdge(nodes["Zurich"], nodes["Berlin"], 85)
        graph.AddEdge(nodes["Berlin"], nodes["Amsterdam"], 85)
        graph.AddEdge(nodes["Berlin"], nodes["Vienna"], 75)
        graph.AddEdge(nodes["Vienna"], nodes["Rome"], 100)
        graph.AddEdge(nodes["Vienna"], nodes["Istanbul"], 130)
        graph.AddEdge(nodes["Warsaw"], nodes["Berlin"], 80)
        graph.AddEdge(nodes["Warsaw"], nodes["Istanbul"], 180)
        graph.AddEdge(nodes["Rome"], nodes["Istanbul"], 155)
        return graph
}
```

Now that you have the graph, take a look at Dijkstra's algorithm:

```
func dijkstra(graph *Graph, city string) {
        visited := make(map[string]bool)
        heap := &Heap{}

        startNode := graph.GetNode(city)
        startNode.value = 0
        heap.Push(startNode)

        for heap.Size() > 0 {
                current := heap.Pop()
                visited[current.name] = true
                edges := graph.Edges[current.name]
                for _, edge := range edges {
```

```
                        if !visited[edge.node.name] {
                                heap.Push(edge.node)
                                if current.value+edge.weight < edge.node.value {
                                        edge.node.value = current.value +
                                        edge.weight
                                        edge.node.through = current
                                }
                        }
                }
        }
}
```

Here is how it works. Let's say you want to find the shortest travel time from London to Istanbul:

1. Set up the weighted graph, a min heap, and a map to mark cities that have been visited before.
2. Get the origin node, London, set its node value to 0, and push it into the heap.
3. The next step is a loop. While the heap is not empty, you pop the heap. This will be your current node.
4. Mark the city as visited.
5. For each city that the current node is connected to (Paris, Luxembourg, and Amsterdam for London), push the city into the heap if it's not been visited before.
6. Check if the current node's value plus the edge's weight is less than the connected node's (Paris, Luxembourg, and Amsterdam) value.
7. If it is, set the connected node's value to the current node's value plus the edge's weight. This is why you set every node (except the originating node, London) to be `MaxInt`.
8. Set the connected node's `through` field to be the current node. The `through` field tells you which node the shortest path to this node is, so you can trace back later to come up with the path.
9. Once you're done, loop from step 3 until all the nodes are visited.

That's it for the algorithm. Here's how you can use it:

```
func main() {
        // build and run Dijkstra's algorithm on graph
        graph := buildGraph()
        city := os.Args[1]
        dijkstra(graph, city)

        // display the nodes
        for _, node := range graph.Nodes {
                fmt.Printf("Shortest time from %s to %s is %d\n",
```

```
                city, node.name, node.value)
            for n := node; n.through != nil; n = n.through {
                fmt.Print(n, " <- ")
            }
            fmt.Println(city)
            fmt.Println()
        }
    }
```

Build the graph and pass it to the algorithm. The results are all in the nodes. After calling the `dijkstra` function, the values of the nodes are now the shortest flight times needed to travel from London while the through fields are the previous nodes that will link back to the shortest path.

If you take the nodes and walk backward to London, you'll see the shortest path:

```
$ % go run dijkstra.go London
Shortest time from London to London is 0
London

Shortest time from London to Paris is 80
Paris <- London

Shortest time from London to Amsterdam is 75
Amsterdam <- London

Shortest time from London to Luxembourg is 75
Luxembourg <- London

Shortest time from London to Zurich is 135
Zurich <- Luxembourg <- London

Shortest time from London to Rome is 205
Rome <- Paris <- London

Shortest time from London to Berlin is 160
Berlin <- Amsterdam <- London

Shortest time from London to Vienna is 215
Vienna <- Zurich <- Luxembourg <- London

Shortest time from London to Warsaw is 240
Warsaw <- Berlin <- Amsterdam <- London

Shortest time from London to Istanbul is 345
Istanbul <- Vienna <- Zurich <- Luxembourg <- London
```

Running Dijkstra's algorithm helps you to find the shortest path from London to all the cities as well!

CHAPTER 15

Image-Processing Recipes

15.0 Introduction

The standard library for 2D image manipulation is the `image` package and the main interface is `image.Image`. To work with the different image formats, you need to register the format first by initializing the format's package in the program's main package:

```
import _ "image/png"
```

Importing a package with an underscore allows you to create the package-level variables and also execute the `init` function in the `image/png` package. You can import more than one format; it doesn't matter and the compiler won't complain (because you're naming it ("_")—if you don't name it, the compiler *will* complain).

Before you start using the `image` package, it's important to know some of its most often-used interfaces and structs.

Image and Other Interfaces

The `image.Image` type is an interface that represents a rectangular grid of `color.Color` pixel values, taken from a color model. This is the main interface for the `image` package. Structs that implement this interface have to implement the `ColorModel`, `Bounds`, and `At` methods:

```
type Image interface {
        ColorModel() color.Model
        Bounds() Rectangle
        At(x, y int) color.Color
}
```

The `color.Color` interface has a method that returns the four values—red, green, blue, and alpha:

```
type Color interface {
    RGBA() (r, g, b, a uint32)
}
```

The `color.Color` interface represents a color. The `At` method in `image.Image` returns the color at the specific location at x, y.

A `image.Rectangle` is a rectangle defined by its top-left and bottom-right points, `Min` and `Max`, respectively:

```
type Rectangle struct {
    Min, Max Point
}
```

And finally, of course, an `image.Point` is a position defined by X and Y values:

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

Image Implementations

Several structs implement `image.Image` in the same package. These include `RGBA`, `NRGBA`, `Gray`, `CMYK`, and `NYCbCrA`, to name a few. I'll focus on `NRGBA` here because it's easy to understand. Let's talk a bit more about what `RGBA` and `NRGBA` mean.

The A in `RGBA` is the alpha channel (also called the image mask) that controls whether parts of the image are visible or not. The `RGBA` image is an image that is alpha-premultiplied or already has the alpha channel applied to it. The N in `NRGBA` means that the alpha channel is not applied to it. In other words, an `NRGBA` image is an image that has an alpha channel but it's not premasked.

The alpha channel is used in a technique called *alpha compositing*, which combines two or more images into a final image called a composite. We won't be doing compositing in this book but if you're compositing images, the difference between having an alpha channel or not having an alpha channel is important. It's also important if you are creating images with transparent backgrounds. Otherwise (meaning if the alpha channel is not used anyway) it doesn't matter.

15.1 Loading an Image from a File

Problem

You want to load an image from an image file.

Solution

Use `image.Decode` to decode data from an image file into an implementation of `image.Image`.

Discussion

If you want to work with an image from a file, you have to open up the file and then decode its content. To do this, you should already know what kind of file you are dealing with and register the type accordingly:

```
func load(filePath string) *image.NRGBA {
        imgFile, err := os.Open(filePath)
        if err != nil {
                log.Println("Cannot read file:", err)
        }
        defer imgFile.Close()

        img, _, err := image.Decode(imgFile)
        if err != nil {
                log.Println("Cannot decode file:", err)
        }
        rimg, ok := img.(*image.NRGBA)
        if ok {
                return rimg, nil
        }
        return nil, errors.New("cannot type assert image")
}
```

First, you need to open a file. Then using `image.Decode`, you decode it into the `img` variable. The `Decode` method returns three values: the first is what you want to get, the `image.Image` value; the second is a string signifying the format of the image; and the third is the usual error value.

Take a closer look at the `img` value. Since you get an `Image` value, why do you want to type assert to `image.NRGBA` before you return it? This is because `Image` is an interface, and `NRGBA` is the actual implementation:

```
type NRGBA struct {
        Pix []uint8
        Stride int
        Rect Rectangle
}
```

In other words, you can manipulate the NRGBA but you can't manipulate an Image. Of course, you can still call methods on Image but you can't directly manipulate it unless you get a hold of the underlying implementation.

15.2 Saving an Image to a File

Problem

You have an image and want to save it to a file.

Solution

Use the Encode method of the correct file format package (e.g., png.Encode for PNG files) to encode the image into a file.

Discussion

If you can load an image from a file you might also want to save it back to a file. Saving images requires you to import the actual format packages because the encoders are specific to the formats (which makes sense if you think about it):

```
import "image/png"
```

If you need to decode and encode, just the preceding code will do; you don't need to import the package and name it with an underscore:

```
func save(filePath string, img *image.NRGBA) {
        imgFile, err := os.Create(filePath)
        defer imgFile.Close()
        if err != nil {
                log.Println("Cannot create file:", err)
        }
        png.Encode(imgFile, img.SubImage(img.Rect))
}
```

First, you create the file. Then use png.Encode (or jpeg.Encode or gif.Encode) to encode the image to the file. You might notice that this example uses the SubImage method in the struct instance. This is because Encode takes in the Image interface as a parameter (as it should). You need to make the NRGBA into an Image so you use the SubImage method and pass in the dimensions of the entire image, which is found in the Rect value.

15.3 Creating Images

Problem

You want to create an image from scratch.

Solution

Create one of the `Image` implementation structs (e.g., `NRGBA`) and populate it with the appropriate data.

Discussion

So you can load and save images from and to a file. What if you want to create an image from scratch? You will have to create an implementation (you can't create an interface). As you remember from earlier, `NRGBA` has three attributes:

`Pix`
: A slice of bytes that contains the pixels in the image (it's just a slice of `color.Color`)

`Stride`
: The distance between the two vertically adjacent pixels

`Rect`
: The dimensions of the image:

```
func main() {
        rect := image.Rect(0, 0, 100, 100)
        img := createRandomImage(rect)
        save("random.png", img)
}

func createRandomImage(rect image.Rectangle) (created *image.NRGBA) {
        pix := make([]uint8, rect.Dx()*rect.Dy()*4)
        rand.Read(pix)
        created = &image.NRGBA{
                Pix:    pix,
                Stride: rect.Dx() * 4,
                Rect:   rect,
        }
        return
}
```

Say you want to create an image that is 100 × 100 pixels. First, you need to create a `Rect` with the correct dimensions. Next, the `Pix` should be a slice of bytes of size 100 × 100 × 4 = 40,000 because each pixel is represented by 4 bytes (R, G, B, and A). Lastly, the `Stride` is the distance between two vertical pixels, which is the width of the image, multiplied by 4, which is 100 × 4 = 400.

The example code created a random image with each pixel a random color by populating the `Pix` slice of bytes with random bytes using `rand.Read`. You can fill it up with anything else, of course.

15.4 Flipping an Image Upside Down

Problem

You want to flip an image upside down.

Solution

Convert an image to a grid of pixels. Swap the positions of the top and bottom pairs of pixels and move down to swap the next pair until all the pixel positions are swapped. Convert the grid of pixels back into a flipped image.

Discussion

In Recipe 15.3 we said the `Image` implementations (for example, `image.NRGBA`) have a slice of bytes that represent pixels in the image. That's not the most intuitive way to represent a raster image. What's more common is a grid of pixels (because a raster image is literally a grid of pixels), so that's the first thing you want to do—convert an `image.NRGBA` to a grid of pixels.

In the `image` package, a pixel is represented by the type `color.Color` so you're going to create a 2D slice of `color.Color` pixels. You will extend the `load` function from before to do this so you can return the grid:

```
func load(filePath string) (grid [][]color.Color) {
    // open the file and decode the contents into an image
        file, err := os.Open(filePath)
        if err != nil {
                log.Println("Cannot read file:", err)
        }
        defer file.Close()
        img, _, err := image.Decode(file)
        if err != nil {
                log.Println("Cannot decode file:", err)
        }
    // create and return a grid of pixels
        size := img.Bounds().Size()
```

```
        for i := 0; i < size.X; i++ {
                var y []color.Color
                for j := 0; j < size.Y; j++ {
                        y = append(y, img.At(i, j))
                }
                grid = append(grid, y)
        }
        return
}
```

As before, you load the file and decode its contents into an image.

To convert the image into a grid, you need to find out the size of the image. Use the `Size` method on the `Rect` struct that is returned from calling the `Bounds` method on the image. This returns a `Point`, which gives you the `X` and `Y` width and length of the image. Iterate the width and length, and at each pixel position, use the `At` method to get the pixel `color.Color`. This will give you a grid of `color.Color` pixels.

You will also need to extend the `save` function from Recipe 15.2 to let you save the grid of pixels back into a file:

```
func save(filePath string, grid [][]color.Color) {
    // create an image and set the pixels using the grid
        xlen, ylen := len(grid), len(grid[0])
        rect := image.Rect(0, 0, xlen, ylen)
        img := image.NewNRGBA(rect)
        for x := 0; x < xlen; x++ {
                for y := 0; y < ylen; y++ {
                        img.Set(x, y, grid[x][y])
                }
        }
    // create a file and encode the image into it
        file, err := os.Create(filePath)
        if err != nil {
                log.Println("Cannot create file:", err)
        }
        defer file.Close()
        png.Encode(file, img.SubImage(img.Rect))
}
```

You will be reversing what you did previously. First, you need to create an image. Using the width and length of the 2D slice, create a `Rect` to represent the size of the image. Create a new image using the `image.NewNRGBA` function, passing it the `Rect`. Then iterate through the grid and at every position, set the `color.Color` pixel from the grid into the new image.

Finally, take the image and encode it into a file.

Now that you can load an image file into a grid of pixels and save it back into an image file, look at the algorithm you want to use to flip the image upside down:

```
func flip(grid [][]color.Color) {
        for x := 0; x < len(grid); x++ {
                col := grid[x]
                for y := 0; y < len(col)/2; y++ {
                        z := len(col) - y - 1
                        col[y], col[z] = col[z], col[y]
                }
        }
}
```

It's relatively simple to flip the image with a grid of pixels. You simply iterate through each column of the grid and swap the top and bottom pixels. Here's how you can use this algorithm to flip the image:

```
func main() {
        grid := load("monalisa.png")
        flip(grid)
        save("flipped.png", grid)
}
```

You will use an image of the Mona Lisa to test (see Figure 15-1).

Figure 15-1. Mona Lisa

First, load the image from a file using `load`, creating a grid of pixels. Then call the `flip` function with the grid. Finally, save the flipped image into another file using `save`. Figure 15-2 shows the Mona Lisa flipped upside down.

Figure 15-2. Flipped Mona Lisa

15.5 Converting an Image to Grayscale

Problem

You want to convert the image to grayscale.

Solution

Convert an image to a grid of pixels. Take each pixel in the grid and convert it to a gray pixel according to the relative luminance formula. Convert the grid of pixels back into an image to get a grayscale image.

Discussion

A grayscale image is an image that has pixels that show different shades of gray representing the amount of light intensity. Black and white represents the opposite ends of the spectrum, with black having the least amount of light and white having the most.

To create a grayscale image from a color image, you can calculate the relative luminance of each pixel from the red, green, and blue values of each pixel. There are a few formulas for calculating this relative luminance, but the simplest is just to take the average of the red, green, and blue values:

```
L = (R + G + B)/3
```

Here's the code to do the conversion:

```
func grayscale(grid [][]color.Color) (grayImg [][]color.Color) {
        xlen, ylen := len(grid), len(grid[0])
        grayImg = make([][]color.Color, xlen)
        for i := 0; i < len(grayImg); i++ {
                grayImg[i] = make([]color.Color, ylen)
        }

        for x := 0; x < xlen; x++ {
                for y := 0; y < ylen; y++ {
                        pix := grid[x][y].(color.NRGBA)
                        gray := uint8(float64(pix.R)/3.0 + float64(pix.G)/3.0 +
                                float64(pix.B)/3.0)
                        grayImg[x][y] = color.NRGBA{gray, gray, gray, pix.A}
                }
        }
        return
}
```

The first part of the `grayscale` function should look familiar; you're creating a new grid of pixels to represent the grayscale image, using the same dimensions as the original image. Next, you iterate through the original image, take each pixel, get the red, green, and blue values, then divide them by 3 and add them up:

```
gray := uint8(float64(pix.R)/3.0 + float64(pix.G)/3.0 + float64(pix.B)/3.0)
```

This is the luminance of the pixel, which you use to create a grayscale pixel that you place in the corresponding position in the grayscale image:

```
grayImg[x][y] = color.NRGBA{gray, gray, gray, pix.A}
```

The luminosity formula here is the simplest, but there are other formulas defined by various standards to convert color to grayscale. One of these standards is the ITU-R BT.709 standard from the International Telecommunications Union (ITU) Radiocommunication Sector, which produces a better result:

```
L = 0.2126 * R + 0.7152 * G + 0.0722 * B
```

Figure 15-3 shows three pictures of a rainbow lorikeet. The one on the left is the colorful original. The middle picture is a grayscale version converted using the BT.709 luminosity formula, while the last picture on the right is a grayscale version converted using the average method.

Figure 15-3. Rainbow lorikeet by David Clode. The first picture on the left (in color if you're reading the web version) is the original. The second picture in the middle is a grayscale version converted using the BT.709 luminosity formula. The last picture on the right is a grayscale version converted using the average method.

15.6 Resizing an Image

Problem

You want to resize an image, making it larger or smaller.

Solution

Convert an image to a grid of pixels as the source and create a new image with the resized dimensions. Use the nearest neighbor interpolation algorithm to figure out the color of each pixel in the new image. Convert the grid of pixels back into a resized image.

Discussion

Resizing a raster image means creating an image with a higher or lower number of pixels. Many algorithms are used for resizing images, including nearest neighbor, bilinear, bicubic, Lanczos resampling, and box sampling. Among these algorithms, the nearest neighbor is probably the simplest. However, the quality is usually not the best, and it can also introduce jaggedness in the image.

The nearest neighbor interpolation algorithm works like this:

1. Assume you are given the original image and the scale of the resize. For example, if you want to double the size of the image, the scale is 2.
2. Create a new image with the new size.
3. Take each pixel in the new image, and divide the X and Y positions by the scale to get `X'` and `Y'` positions. These might not be whole numbers.
4. Find the floor of `X'` and `Y'`. These are the corresponding X and Y positions mapped on the original image.

5. Use the X' and Y' positions to get the pixel in the original image and put it into the new image at the X and Y positions.

Here's the code to implement the algorithm:

```
func resize(grid [][]color.Color, scale float64) (resized [][]color.Color) {
        xlen, ylen := int(float64(len(grid))*scale), int(float64(len(grid[0]))*
        scale)
        resized = make([][]color.Color, xlen)
        for i := 0; i < len(resized); i++ {
                resized[i] = make([]color.Color, ylen)
        }
        for x := 0; x < xlen; x++ {
                for y := 0; y < ylen; y++ {
                        xp := int(math.Floor(float64(x) / scale))
                        yp := int(math.Floor(float64(y) / scale))
                        resized[x][y] = grid[xp][yp]
                }
        }
        return
}
```

As you can see, it's quite straightforward. Per the algorithm, you create a new resized image, then iterate through each pixel in the new image and find the corresponding matching pixel in the original image. After that, you copy that pixel from the original image over to the new image.

Figure 15-4 shows the Mona Lisa resized. The picture on the left is the original image, the picture in the middle is a 10× size-reduced image, and the picture on the right is a 10× size-enlarged image.

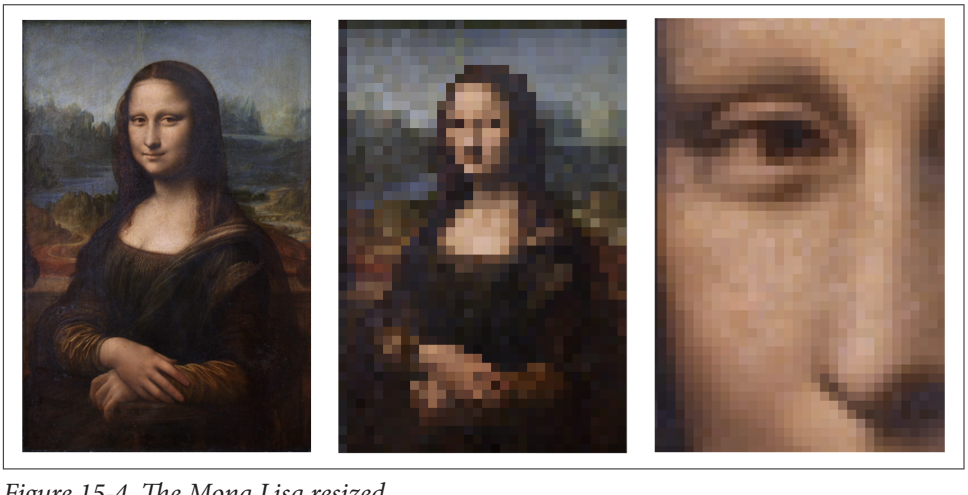

Figure 15-4. The Mona Lisa resized

CHAPTER 16

Networking Recipes

16.0 Introduction

While individual computers are powerful in their own right in processing data and computing results, the real power of computers lies in connecting them to a computer network. When computers are networked, they can tap into each other's strengths and capabilities, aggregate computing power, and distribute processing among each other such that complex problems can be split up and worked on separately.

Computer networks use network protocols to communicate with each other. Network protocols are often abstracted into different layers. For example, Open Systems Interconnection (OSI) describes seven layers of communication protocols—starting from the application layer at the top, followed by the presentation, session, transport, network, data link, and physical layers. The Internet Protocol suite, popularly known as TCP/IP, has only four layers, starting with the application layer, followed by the transport, internet, and link layers. TCP/IP predates OSI and is more commonly used, but either protocol suite is only an abstraction layer used to describe the network communications.

The application layer is, as the name suggests, the protocol layer that describes how applications talk to each other. Examples of protocols on this layer include HyperText Transfer Protocol (HTTP) and File Transfer Protocol (FTP).

The transport layer describes how datagrams are sent and received. The two main protocols in this layer are Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). TCP ensures packets of data (called datagrams) are delivered reliably. UDP, on the other hand, doesn't ensure datagram delivery. TCP is reliable but has a high overhead, while UDP is often much faster.

Internet layer protocols describe how bits and bytes of data are organized into datagrams and how devices on the network find each other. The Internet Protocol (IP) is the most widely used internet layer protocol in the world. Two main versions of IP are in use today: the first is IP version 4, which is identified by a 4-byte number written in a dotted quad format—for example, 192.168.1.1—where each of the four numbers is an unsigned byte with value from 0 to 255.

There are about 4 billion possible IPv4 addresses, which means they are running out. IP version 6, on the other hand, has a 16-byte address that provides 340 undecillion (340 trillion trillion trillion) addresses, which should be more than enough for a while. IPv6 addresses are written in eight blocks of four hexadecimal digits separated by colons—for example, 2001:0db8:85a3:0000:0000:8a2e:0370:7334.

In addition to addresses, each IP address has 65,535 logical ports, which are normally used to identify a service. For example, HTTP normally uses port 80 while FTP uses two ports: port 20 for command and port 21 for data. Port numbers 1 to 1023 are reserved for well-known services such as HTTP or FTP.

Go, like many other programming languages, provides standard libraries for network programming. In particular, the `net` package in the standard library provides capabilities for socket programming. This chapter focuses on socket programming.

You will be using `netcat` to test our network programs. `netcat` is a simple utility that can be used to read and write data across network connections, using TCP or UDP. It is often used to test network connectivity and port availability.

In Unix-like operating systems, `netcat` is usually installed by default. On macOS, instead of `netcat` you can use `nc`, which works roughly the same way (also installed by default). In Windows, download `netcat` from Nmap.

In this chapter, you will use `nc` as the command name.

16.1 Creating a TCP Server

Problem

You want to create a TCP server to receive data from a TCP client.

Solution

Use the `Listen` function in the `net` package to listen for connections, then accept the connection using `Accept`.

Discussion

TCP is a connection-oriented protocol at the transport layer. It ensures a reliable and ordered delivery of bidirectional data by managing message acknowledgments and sequences of data packets. As a result, it's more reliable. When a TCP connection is established, it is maintained until the applications on both ends finish exchanging messages and close it.

Sockets are application-level connections between two computers and represent endpoints for sending and receiving data to other programs across the network. Sockets abstract the complexities behind networking, allowing programmers to develop programs that communicate through the network. As a result, writing network programs are often about socket programming.

There are three parts to a simple TCP server program:

1. Listen for incoming connections.
2. Accept the connection.
3. Read and optionally write data to the connection.

Here is the code:

```
func main() {
        listener, err := net.Listen("tcp", "localhost:9000")
        if err != nil {
                log.Fatal(err)
        }
        defer listener.Close()
        for {
                conn, err := listener.Accept()
                if err != nil {
                        log.Fatal(err)
                }

                go func(c net.Conn) {
                        buf := make([]byte, 1024)
                        _, err := c.Read(buf)
                        if err != nil {
                                log.Fatal()
                        }
                        log.Print(string(buf))
                        conn.Write([]byte("Hello from TCP server"))
                        c.Close()
                }(conn)
        }
}
```

First, you set up the server to listen to a socket using the `net.Listen` function. Sockets are identified by a combination of the transport protocol, IP address, and port number. Then you loop indefinitely to accept connections. When a connection comes, it is accepted and handled in a separate goroutine. The goroutine reads the data from the connection and prints it out. The connection is then closed.

The `net.Listen` function returns a `net.Listener` interface, a generic network listener for stream-oriented protocols. The `net.Listen` function takes two arguments. The first is the network protocol, which is `tcp` in this case. The second is the address to listen on, which is in the form `<host>:<port>`. If the host is provided, the listener will listen only to that IP address. If it's left empty, as in this case, for example, `:9000`, it will listen on all available unicast and anycast IP addresses of the local system. Interestingly if it's a single hostname it will also listen only for IPv4 traffic. If you leave it empty it will listen to both IPv4 and IPv6. If the port is 0, a random port is chosen and the `Addr` method of `net.Listener` can be used to retrieve the port number.

You read from the connection using the `Read` method. The `Read` method takes a byte slice as an argument and returns the number of bytes read and an error. The byte slice is used to store the data read from the connection.

You also write to the connection using the `Write` method. The `Write` method takes a byte slice as an argument and returns the number of bytes written and an error.

Here's how this works. Start the server first:

```
$ go run main.go
```

Then you can use the `nc` (netcat) command to connect and send data to the server. In another terminal, run the following command as the client:

```
$ echo "Hello from TCP client" | nc localhost 9000
Hello from TCP server
```

Echo the string "Hello from TCP client" to the `nc` command, which sends it to the server. The server then prints out the string and sends back "Hello from TCP server" to the client.

"Hello from TCP server" prints out on the client side.

You might wonder what happens if you have more than 1,024 bytes from the client. You can look until `io.EOF` is reached. Here is the snippet:

```
go func(c net.Conn) {
        bytes := []byte{}
        for {
                buf := make([]byte, 32)
                _, err := c.Read(buf)
                if err != nil {
```

```
                        if err == io.EOF {
                                break
                        } else {
                                log.Fatal(err)
                        }
                }
                bytes = append(bytes, buf...)
        }
        log.Print(string(bytes))
        _, err = conn.Write([]byte("Hello from TCP server"))
        if err != nil {
                log.Fatal(err)
        }
        c.Close()
}(conn)
```

You will loop until all the data from the client is read (and therefore `io.EOF` is encountered).

You're sending data from the `nc` client using IPv4. If you want to use IPv6, you can use the `-6` flag at the client but you also need to change the listener:

```
listener, err := net.Listen("tcp", ":9000")
```

If you do this, you can use the `nc` command to connect to the server using IPv6:

```
$ echo "Hello from TCP client" | nc -6 localhost 9000
Hello from TCP server
```

The `net.Conn` interface returned by the `Accept` method of `net.Listener` represents a generic stream-oriented network connection. It is an abstraction of a network connection and has `Read` and `Write` methods, meaning it is both a `Reader` and `Writer`.

Both `net.Listener` and `net.Conn` are interfaces, and in this case, they are implemented by the `net.TCPListener` and the `netTCPConn` structs, respectively. Instead of using the `net.Listen` and `Accept` functions, you could have created the `net.TCPListener` and `net.TCPConn` structs directly using the `net.ListenTCP` and `AcceptTCP` functions. However, the `net.Listen` and `Accept` functions are more convenient and portable:

```
func main() {
        addr, err := net.ResolveTCPAddr("tcp", ":9000")
        if err != nil {
                log.Fatal(err)
        }
        listener, err := net.ListenTCP("tcp", addr)
        if err != nil {
                log.Fatal(err)
        }
        defer listener.Close()
```

```
        for {
                conn, err := listener.AcceptTCP()
                if err != nil {
                        log.Fatal(err)
                }
                go func(c net.Conn) {
                        buf := make([]byte, 1024)
                        _, err := c.Read(buf)
                        if err != nil {
                                log.Fatal()
                        }
                        log.Print(string(buf))
                        conn.Write([]byte("Hello from TCP server"))
                        c.Close()
                }(conn)
        }
}
```

It might seem redundant that you have more than one way of creating a TCP server. The `net.Listen` and `Accept` functions are more convenient and simpler to use. `net.ListenTCP` and `AcceptTCP` are more verbose but give you more control over the connection; for example, you could set the `KeepAlive` property of the connection to true to keep the TCP connection alive longer.

16.2 Creating a TCP Client

Problem

You want to create a TCP client to send data to a TCP server.

Solution

Use the `Dial` function in the `net` package to connect to a TCP server.

Discussion

Creating a TCP client is similar to creating a TCP server but even simpler. The main difference is that the client connects to a server instead of listening for connections:

```
func main() {
        conn, err := net.Dial("tcp", ":9000")
        if err != nil {
                log.Fatal(err)
        }
        defer conn.Close()
        conn.Write([]byte("Hello World from TCP client"))
}
```

First, connect to the server using the `net.Dial` function. Then write data to the connection; when you're done you can close the connection.

You saw the `net.Conn` interface in Recipe 16.1. The `net.Dial` function returns a `net.Conn` interface, which is a generic stream-oriented network connection. It is an abstraction of a network connection and has `Read` and `Write` methods, meaning it is both a `Reader` and `Writer`. In a TCP client, you will use this connection to write data to the server.

Use `nc` to listen on a port and see what the client sends:

```
$ nc -l 9000
```

Then run the client on another terminal:

```
$ go run main.go
```

On the `nc` terminal "Hello World from TCP client" prints out.

How about IPv6? You can use the `-6` flag to force `nc` to listen on IPv6:

```
$ nc -l -6 9000
```

If you run the client again you will see the same output. Go TCP clients send out the data using both IPv4 and IPv6.

Just as you can create a TCP server using the `net.ListenTCP` and `AcceptTCP` functions, you can also create a TCP client using the `net.DialTCP` function:

```
func main() {
        addr, err := net.ResolveTCPAddr("tcp", ":9000")
        if err != nil {
                log.Fatal(err)
        }
        conn, err := net.DialTCP("tcp", nil, addr)
        if err != nil {
                log.Fatal(err)
        }
        defer conn.Close()
        conn.Write([]byte("Hello World from TCP Client"))
}
```

The `net.DialTCP` function takes a network string and two `net.TCPAddr` structs as arguments. You can create the address structs using the `net.ResolveTCPAddr` function. The first argument to `net.DialTCP` is the network, followed by the local address and the remote address. If the local address is nil, a local address is automatically chosen. If the remote address is nil, an error is returned (the documentation says otherwise but as of Go 1.20, in the *net/tcpsock.go* source file, this is the behavior).

16.3 Creating a UDP Server

Problem

You want to create a UDP server to receive data from a UDP client.

Solution

Use the `ListenPacket` function in the `net` package to listen for incoming packets. Then use the `ReadFrom` method of the `PacketConn` interface to read data from the connection. You can also use the `WriteTo` method to write data to the connection.

Discussion

UDP is a connectionless protocol. This means there is no connection between the client and the server. The client sends data to the server and the server sends data back to the client. Neither the client nor the server knows if the server or client received the data or not.

A UDP server in Go is similar to a TCP server. The main difference is that you use the `net.ListenPacket` function instead of the `net.Listen` function:

```
func main() {
        conn, err := net.ListenPacket("udp", ":9001")
        if err != nil {
                log.Fatal(err)
        }
        defer conn.Close()
        buf := make([]byte, 1024)
        for {
                _, addr, err := conn.ReadFrom(buf)
                if err != nil {
                        log.Fatal(err)
                }
                log.Printf("Received %s from %s", string(buf), addr)
                conn.WriteTo([]byte("Hello from UDP server"), addr)
        }
}
```

The `net.ListenPacket` function returns a `net.PacketConn` interface, which is a packet-oriented network connection. Unlike the `net.Conn` interface, `net.PacketConn` is not a `Reader` or `Writer` because it doesn't have `Read` or `Write` methods. Instead, it has `ReadFrom` and `WriteTo` methods to read and write data from and to the connection.

You can test the server. Start this on one terminal:

```
$ go run main.go
```

Use the `nc` command to send data to it. In another terminal, run the following command as the client:

```
$ echo "Hello from UDP client" | nc -u localhost 9001
```

You should see "Received Hello from UDP client" printed out on the server side, and "Hello from UDP server" printed out on the client side.

As in the TCP server, your UDP server is listening on both IPv4 and IPv6. To show this you'll use the `-6` flag to force `nc` to use IPv6:

```
$ echo "Hello from UDP client" | nc -u -6 localhost 9001
```

You should see the same output as before.

Besides using `net.ListenPacket` you can also use the `net.ListenUDP` function to create a UDP server:

```
func main() {
        addr, err := net.ResolveUDPAddr("udp", ":9001")
        if err != nil {
                log.Fatal(err)
        }
        conn, err := net.ListenUDP("udp", addr)
        if err != nil {
                log.Fatal(err)
        }
        defer conn.Close()
        buf := make([]byte, 1024)
        for {
                _, addr, err := conn.ReadFromUDP(buf)
                if err != nil {
                        log.Fatal(err)
                }
                log.Printf("Received %s from %s", string(buf), addr)
                conn.WriteToUDP([]byte("Hello from UDP server"), addr)
        }
}
```

Unlike the `net.ListenPacket`, though, the `net.ListenUDP` function takes a `net.UDPAddr` as an argument. First, use the `net.ResolveUDPAddr` function to resolve and create the address. Then use the `net.ListenUDP` function to create the connection. Instead of `net.PacketConn` you get a `net.UDPConn` interface. The `net.UDPConn` struct implements both the `net.PacketConn` and `net.Conn` interfaces, meaning it has both `ReadFrom` and `WriteTo` methods and `Read` and `Write` methods. In addition, it has `ReadFromUDP` and `WriteToUDP` methods to read and write data from and to the connection.

The `net.ListenPacket` function is more generic and can be used to create a UDP server, but it is also used to create other types of servers such as IP and Unix domain

sockets. The `net.ListenUDP` function is more specific and can only be used to create a UDP server.

At this point, you might be confused about why `net.UDPConn` implements both the `net.PacketConn` and `net.Conn` interfaces. After all, `net.UDPConn` is a packet-oriented connection whereas `net.Conn` is a stream-oriented connection, which is also implemented by the `net.TCPConn` struct. How can a connection be both packet-oriented and stream-oriented at the same time?

This comes from the original Berkeley, sockets API that has become the standard in Unix-like systems where sockets are stream-oriented. Both UDP and TCP are implemented on top of sockets so they both implement the `net.Conn` interface.

16.4 Creating a UDP Client

Problem

You want to create a UDP client to send data to a UDP server.

Solution

Use the `Dial` function in the `net` package to connect to a UDP server. Then use the `Write` method of the `net.UDPConn` interface to write data to the connection.

Discussion

Creating a UDP client can be very straightforward, and it can look exactly like the TCP client, except the network string is `udp` instead of `tcp`:

```
func main() {
        conn, err := net.Dial("udp", ":9001")
        if err != nil {
                log.Fatal(err)
        }
        defer conn.Close()
        conn.Write([]byte("Hello from UDP client"))
}
```

Try it out: set up a UDP listener using `nc` on one terminal. Use the `-u` flag to force `nc` to use UDP and the `-l` flag to listen for incoming connections:

```
$ nc -l -u 9001
```

Then run your UDP client on another terminal. You should see "Hello from UDP client" printed out on the server side.

This works for IPv4 and IPv6. To test this, you'll use the `-6` flag to force `nc` to use IPv6 on the server side:

```
$ nc -l -u -6 9001
```

If you run the same client you should see the same output.

You can also use the `net.DialUDP` function to create a UDP client:

```
func main() {
        raddr, err := net.ResolveUDPAddr("udp", ":9001")
        if err != nil {
                log.Fatal(err)
        }
        conn, err := net.DialUDP("udp", nil, raddr)
        if err != nil {
                log.Fatal(err)
        }
        defer conn.Close()

        _, err = conn.Write([]byte("Hello from UDP client"))
        if err != nil {
                log.Fatal(err)
        }
}
```

The `net.DialUDP` function takes a `net.UDPAddr` as argument. First, use the `net.ResolveUDPAddr` function to resolve and create the address. Then use the `net.DialUDP` function to create the connection.

To write to the connection use the `Write` method of the `net.UDPConn` interface. The `Write` method takes a byte slice as an argument and returns the number of bytes written and an error.

CHAPTER 17

Web Recipes

17.0 Introduction

Web applications are everywhere. Take any software application you use daily, and it is likely a web application. Any programming language that supports developing software that interfaces with human beings will inevitably support developing web applications as well. One of the first libraries and frameworks to be built for any new language is interaction with the internet and the World Wide Web. Go is no different.

A web application is a computer program that responds to an HTTP request by a client and sends back HTML to the client in an HTTP response. In other words, a web application is a server—a web server, to be exact. The client is usually a web browser, and they communicate over HTTP.

A web service, on the other hand, is a computer program that responds to an HTTP request by a client that is not a browser used by a human user but another computer program. A web service is a server as well, but it usually returns JSON, and it increasingly also returns binary formats.

HTTP is the application-level communications protocol that powers the World Wide Web. Everything that you see on a web page is transported through this seemingly simple text-based protocol. HTTP is simple but surprisingly powerful—since its definition in 1990, it has gone through only three iterative changes. HTTP 1.1 is the most widely used version, while HTTP 2 is the current version. HTTP 3 is in the works.

This chapter will explore the Go standard library and the Go programming language's support for web application development.

Parts of a Web Application

A web application generally consists of three parts:

A multiplexer
: A router that matches the request URI to a handler function

One or more handlers
: Functions that handle the requests and return the responses

A template engine
: An engine that combines one or more templates with data and renders the response

The multiplexer is quite straightforward. It simply matches the request URI to a handler according to a URL route. For example, you want to match the URL route `/home` to a `homeHandler` function.

The handler function is where the real work is done. It takes in a request, does some processing with the data from the request, and returns a response.

The template engine is used to render the body of the response. It combines one or more templates with data and returns the response body. While this is commonly HTML, it can be any format, including JSON, XML, plain text, or even binary data, such as images and PDFs.

Now that you've had a quick introduction, you can learn more about how to use Go in web development.

17.1 Creating a Simple Web Application

Problem

You want to create a simple web application that responds to an HTTP request and sends back an HTTP response.

Solution

Use the `net/http` package to create a simple Hello World web application.

Discussion

You want to create a simple web application running on port 8000 that shows a Hello World message when a user visits the `/` URL.

The only package you need is the `net/http` package:

```
package main

import (
        "net/http"
)

func main() {
        http.HandleFunc("/", index)
        http.ListenAndServe(":8000", nil)
}

func index(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("Hello World"))
}
```

Starting from the second line in the `main` function: here, you call the `http.ListenAndServe` function to start the web server. The first parameter is the address to listen on, which is in the format `<host>:<port>`. If the host is provided, the listener will listen only to that IP address. If it's left empty, as in this case, for example, `:8000`, it will listen on all available unicast and anycast IP addresses of the local system.

The second parameter is a handler; that is, it is a struct that implements the `http.Handler` interface. The `http.Handler` interface has only one method, `ServeHTTP`, which takes in an `http.ResponseWriter` and an `http.Request` as parameters. The `http.ResponseWriter` is used to write the response back to the client, and the `http.Request` contains all the information about the request.

You might be wondering why you are setting up the server to use only one handler; you would expect a lot more than just one handler. This is because in Go, the multiplexer, the router that matches the request URI to a handler, is a handler itself!

In this case, it's left as nil, which means it will assume the default multiplexer, `http.DefaultServeMux`. `http.DefaultServeMux` is an instance of the `http.ServeMux` struct, which in turn, implements the `http.Handler` interface. You don't normally interact directly with `http.DefaultServeMux`, despite it being a variable, but many of the functions in the package are nothing more than wrappers around it, as you will see in a while.

The `http.ServeMux` multiplexer has a `HandleFunc` method that registers a function as a handler for a given URL pattern. This `HandleFunc` method takes in a URL pattern and a function as parameters. The function must have the signature `func(http.ResponseWriter, *http.Request)` and will be called when a request matches the URL pattern.

Now when you look at the first line of the main function, things should become clear. You call the `http.HandleFunc` function to register the `index` function as the handler

for the / URL pattern. Internally what it does is simply call the `HandleFunc` method of the `http.DefaultServeMux` multiplexer.

The `index` function is quite trivial. In this case, you simply write the string `Hello World` to the `http.ResponseWriter`. It might not look like it, but `w` is a pointer. `http.ResponseWriter` is an interface so you can't tell if it's a pointer or not. Whatever is written to the `http.ResponseWriter` will then be sent back to the client as the response body.

Start the server:

```
$ go run main.go
```

Now visit *http://localhost:8000/* in your browser. You should see the Hello World message displayed, as shown in Figure 17-1.

Figure 17-1. Hello World web application

In this recipe, you used the multiplexer in the standard library because it is the default implementation. It is quite common to use more sophisticated and performant multiplexers offered by third-party packages; for example, the `chi` package.

Here's how the code would look with a third-party multiplexer like chi:

```
package main

import (
        "net/http"

        "github.com/go-chi/chi/v5"
)

func main() {
        mux := chi.NewRouter()
        mux.Get("/", index)
        http.ListenAndServe(":3000", mux)
}

func index(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("Hello World"))
}
```

As you can see, the main change is that you need to create a new multiplexer, `mux`, and this will be used instead of the default multiplexer when you start the server with `http.ListenAndServe`. Third-party multiplexers are more optimal and capable. In

most cases, you should use one unless you are writing something that only uses the standard library.

17.2 Handling HTTP Requests

Problem

You want to process HTTP requests and send back HTTP responses.

Solution

Use `http.Request` to extract information on HTTP requests and `http.ResponseWriter` to send HTTP responses back.

Discussion

In the previous recipe, you created a simple web application that responds to an HTTP request and sends back an HTTP response. In this recipe, we will look deeper into how to extract data from the HTTP requests.

The `http.Request` struct represents an HTTP request message sent from the client. The struct contains important information about the request, as well as several useful methods. Some important parts of `Request` are:

- URL
- Header
- Body
- Form, PostForm, and MultipartForm

You can also get access to the cookies in the request and the referring URL, and the user agent from methods in `Request`.

The `URL` field is a representation of the URL sent as part of the request line (the first line of the HTTP request). The `URL` field is a pointer to the `url.URL` type.

The URL will look like this:

```
scheme://[userinfo@]host/path[?query]
```

The two most commonly used pieces of information from the URL of a request are the path and the query. The `path` is the path to the resource. The `query` is the query string, which often has the form of a key-value pair, for example, `?key=value`.

You can also get the HTTP method as well as the hostname. The HTTP method is the method used to make the request, such as `GET`, `POST`, `PUT`, `DELETE`, and so on. The hostname is the name of the server that the request was sent to.

Here's a quick look at this:

```
func main() {
        http.HandleFunc("/hello/world", hello)
        http.ListenAndServe(":8000", nil)
}

func hello(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintf(w, "Method : %s, Host : %s", r.Method, r.Host)
        fmt.Fprintf(w, "Path : %s, Query : %s\n", r.URL.Path, r.URL.Query())
}
```

Run this on the server and open *http://localhost:8000/hello/world?name=sausheong*. Figure 17-2 shows a screenshot of a browser showing the method and host of the request, as well as the path and query of the URL.

Figure 17-2. Method, host, path, and query

As you can see, the `Query` field is a map of key-value pairs.

Next are the request headers. The `Header` field of an `http.Request` is a map of all the HTTP headers sent with the request. The keys in the map are the header names and the values in the map are slices of strings, so if a header appears multiple times in the request, the values will be appended to the slice:

```
func main() {
        http.HandleFunc("/headers", headers)
        http.ListenAndServe(":8000", nil)
}

func headers(w http.ResponseWriter, r *http.Request) {
        for k, v := range r.Header {
                fmt.Fprintf(w, "%s: %s\n", k, v)
        }
}
```

Run this on the server and open *http://localhost:8000/headers*. Figure 17-3 is a screenshot of a browser showing the request headers.

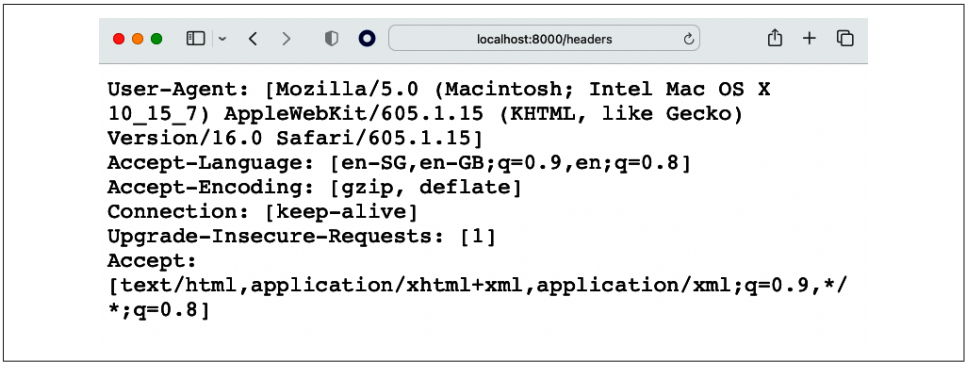

Figure 17-3. Request headers

Don't be overly taken by the headers. These are simply the headers that are sent by the browser. In this case, I'm using Safari, so these are the headers sent by Safari. If you use a different browser, you will see different headers.

Next is the body of a request. The `Body` field of an `http.Request` is an `io.ReadCloser` that contains the request body. The request body is available only if the request has a body, such as a `POST` request:

```
func main() {
        http.HandleFunc("/body", body)
        http.ListenAndServe(":8000", nil)
}

func body(w http.ResponseWriter, r *http.Request) {
        body, _ := io.ReadAll(r.Body)
        fmt.Fprintf(w, "%s", body)
}
```

Since the `Body` field is an `io.ReadCloser`, you can use the `io.ReadAll` function to read the entire body into a byte slice. To test this, you can use the `curl` command:

```
$ curl -X POST -d "Hello World" http://localhost:8000/body
```

The `curl` command will send a `POST` request with the body `Hello World` to the server. The server will then send back the body of the request. If you run this on the command line, you will see the following:

```
 % curl -X POST -d "Hello World" http://localhost:8000/body
Hello World
```

17.3 Handling HTML Forms

Problem

You want to process data submitted from HTML forms.

Solution

Use the `Form` field of `http.Request` or the `FormValue` method to access the data submitted from HTML forms.

Discussion

Before we dive into handling form data from a `POST` request, let's take a closer look at HTML forms. A typical HTML form often looks like this:

```
<form action="/process" method="post">
  <input type="text" name="name"/>
  <input type="text" name="book"/>
  <input type="submit"/>
</form>
```

Within the `<form>` tag, you place several HTML form elements including text input, text area, radio buttons, and so on. These elements allow users to enter data to be submitted to the server. Data is submitted to the server when the user clicks a submit button or somehow triggers the form submission.

The HTML form data is always sent as name-value pairs. The format of the name-value pairs sent through a `POST` request is specified by the content type of the HTML form. This is defined using the `enctype` attribute:

```
<form action="/process" method="post"
enctype="application/x-www-form-urlencoded">
  <input type="text" name="name"/>
  <input type="text" name="book"/>
  <input type="submit"/>
</form>
```

The default value for `enctype` is `application/x-www-form-urlencoded`. This means our HTML forms don't normally need to specify the `enctype`. Browsers are required to support at least `application/x-www-form-urlencoded` and `multipart/form-data`. If you set `enctype` to `application/x-www-form-urlencoded`, the browser will encode a long query string in the HTML form data, with the name-value pairs separated by an ampersand (&) and the name separated from the values by an equals sign (=). That's the same as URL encoding, hence the name. In other words, the HTTP body will look something like this:

```
name=sau%20sheong&book=go%20cookbook
```

If you set `enctype` to `multipart/form-data`, each name-value pair will be converted into a MIME message part, each with its own content type and content disposition. When would you use one or the other? If you're sending simple text data, the URL-encoded form is better—it's simpler and less processing is needed. If you're sending large amounts of data, the `multipart/form-data` form is better.

Now that you know how HTML forms work, take a look at how Go handles HTML forms:

```
func main() {
        http.HandleFunc("/form", form)
        http.ListenAndServe(":8000", nil)
}

func form(w http.ResponseWriter, r *http.Request) {
        r.ParseForm()
        for k, v := range r.Form {
                fmt.Fprintf(w, "%s: %s\n", k, v)
        }
}
```

Handling HTML forms in Go is pretty straightforward. The `Form` field of an `http.Request` is a map of all the form data sent with the request. The keys in the map are the form element names and the values in the map are slices of strings. However, before you start accessing the `Form` field you need to call the `ParseForm` method on the `http.Request` to parse the form data. If you don't call `ParseForm`, the `Form` field will be nil.

Take a look at this in action. You can use the `curl` command to send a `POST` request with the form data to the server:

```
$ curl -X POST -d "name=sau sheong&book=go cookbook" http://localhost:8000/form
```

If you run this on the command line, you will see the following:

```
name: [sau sheong]
book: [go cookbook]
```

If you know exactly what you want from the form, you can actually get the form data even faster. The `FormValue` method is a convenience method that returns the first value for the named component of the query. If no values are associated with the key, it returns the empty string. If multiple values are associated with the key, it returns the first value:

```
func main() {
        http.HandleFunc("/form_value", formValue)
        http.ListenAndServe(":8000", nil)
}

func formValue(w http.ResponseWriter, r *http.Request) {
```

```
        fmt.Fprintln(w, r.FormValue("name"))
}
```

If you run the same `curl` command again, you will see the following:

```
sau sheong
```

You might notice that you don't even need to parse the form data to get the form value. This is because the `FormValue` method will automatically parse the form data for you.

17.4 Uploading a File to a Web Application

Problem

You want to upload a file to a web application.

Solution

Use the `net/http` package to create a web application and the `io` package to read the file.

Discussion

Submitting a file to a web application is a common task. For example, you might want to upload a profile picture to a social network, or you might want to upload a file to a file-sharing service. Uploading files is commonly done through HTML forms. You learned that HTML forms have an `enctype` attribute that specifies the format of the form data. By default, the `enctype` is set to `application/x-www-form-urlencoded`. This means that the form data is encoded as a query string. However, if you set the `enctype` to `multipart/form-data`, the form data will be encoded as a MIME message. This is the format you need to upload files.

Here's an example. You will create a web application that allows users to upload a file. The web application will display the filename and the file size. The web application will also save the file to the local filesystem:

```
func main() {
        http.HandleFunc("/upload", upload)
        http.ListenAndServe(":8000", nil)
}

func upload(w http.ResponseWriter, r *http.Request) {
        file, fileHeader, err := r.FormFile("uploadfile")
        if err != nil {
                fmt.Println(err)
                return
        }
```

```
        defer file.Close()
        fmt.Fprintf(w, "%v", fileHeader.Header)
        f, err := os.OpenFile("./uploaded/"+fileHeader.Filename,
                os.O_WRONLY|os.O_CREATE, 0666)
        if err != nil {
                fmt.Println(err)
                return
        }
        defer f.Close()
        io.Copy(f, file)
}
```

As usual, the action is at the handler function. The first thing you need to do is to get the file from the HTML form. You can do this using the `FormFile` method. The `FormFile` method takes the name of the file input element as its argument and returns two values and an error. The first value is the file, an `io.ReadCloser`, and the second value is the file metadata, a `multipart.FileHeader`.

Using the file header, get the filename and then create a new file. Then use the `io.Copy` function to copy the file from the `io.ReadCloser` to the new file.

To test this, use `curl` to post a file to the server form. The syntax for `curl` is to use the `-F` (form) option, which will add `enctype="multipart/form-data"` to the request. The argument to this option is a string with the name of the file form field (`uploadfile`), followed by `=` and then `@` followed by the path to the file to upload:

```
$ curl -F "uploadfile=@lenna.png" http://localhost:8000/upload
```

Once you run this command, you should see a file *lenna.png* created in the `.uploaded` directory.

17.5 Serving Static Files

Problem

You want to serve static files such as images, CSS, and JavaScript files.

Solution

Use the `http.FileServer` function to serve static files.

Discussion

Web applications often need to serve static files such as images, CSS, and JavaScript files. The `net/http` package provides a `FileServer` function that can be used to serve static files.

You can break it down into parts to analyze it better:

```
func main() {
        dir := http.Dir("./static")
        fs := http.FileServer(dir)
        http.Handle("/", fs)
        http.ListenAndServe(":8000", nil)
}
```

First, you need to understand where you want to serve the files from. For this, use `http.Dir`. Despite how it looks, `http.Dir` is not a function but a type. It's a type that implements the `http.FileSystem` interface so the first line typecasts the directory such that it becomes a `http.Dir`.

Next, the `http.FileServer` function takes in the `http.Dir` as a parameter and returns an `http.Handler` that can be used to serve the files.

Finally, call the `http.Handle` function to register the `http.Handler` as the handler for the `/` URL pattern.

The previous information broke down the steps, but most of the time, you'll use it this way:

```
func main() {
        http.Handle("/", http.FileServer(http.Dir("./static")))
        http.ListenAndServe(":8000", nil)
}
```

If you go to the URL from the browser, you should see the screen in Figure 17-4, which is a screenshot of a browser showing the file that was served.

This works because you use `/` as the URL to start from. However, once you start using different ones, you'll need to use `http.StripPrefix` to remove the prefix:

```
func main() {
        dir := http.Dir("./static")
        fs := http.FileServer(dir)
        fs = http.StripPrefix("/static", fs)
        http.Handle("/static/", fs)
        http.ListenAndServe(":8000", nil)
}
```

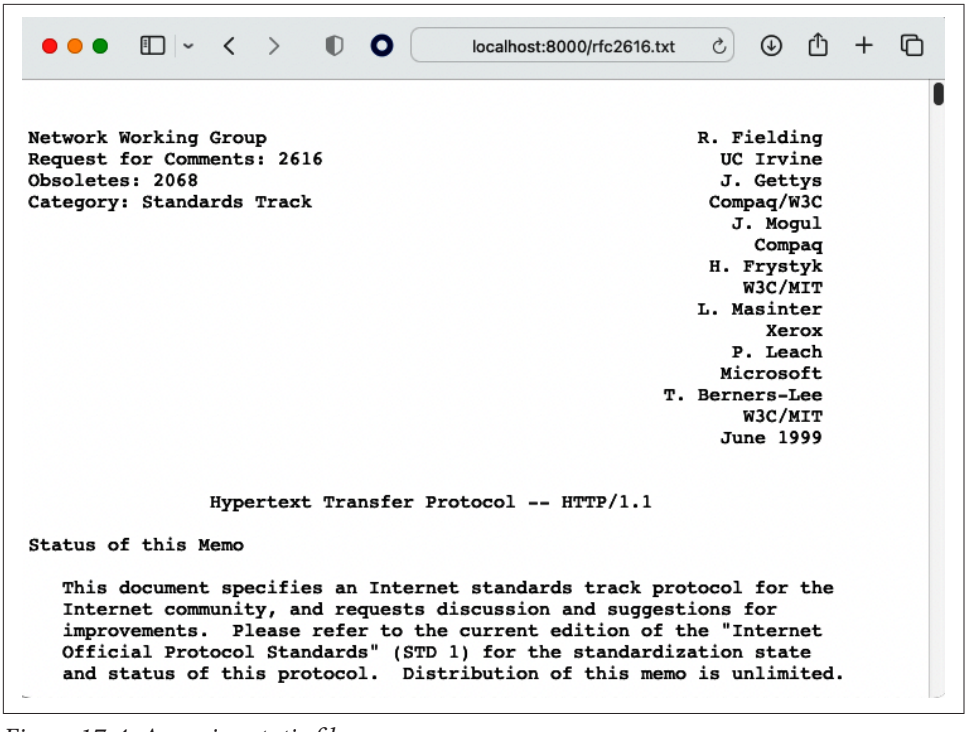

Figure 17-4. Accessing static files

The main difference comes when you want to use `/static` as the root URL (or any other starting point). If you proceed as before, when the user requests for a file */static/rfc2616.txt*, the file server you look for a file called */static/static/rfc2616.txt*. To avoid this, use the `http.StripPrefix` function to remove the prefix `/static` from the request URL before it is passed to the file server. As before, you normally do it all in a single line:

```
func main() {
        http.Handle("/static/", http.StripPrefix("/static",
                http.FileServer(http.Dir("./static"))))
        http.ListenAndServe(":8000", nil)
}
```

This seems quite redundant, but actually there is another reason for this. In the example, you have a directory called `static`, and you want to serve it out from the `/static` URL. You could have a directory called `datafiles`, for example, but you want the user to access it from the `/static` URL:

```
func main() {
        http.Handle("/static/", http.StripPrefix("/static",
                http.FileServer(http.Dir("./datafiles"))))
        http.ListenAndServe(":8000", nil)
}
```

In other words, while *rfc2616.txt* is in the `datafiles` directory, the user accesses it from the URL `/static/rfc2616.txt`.

If you go to the URL from the browser, you should see the screen in Figure 17-5, which is a screenshot of a browser showing the file that was served from the "static" URL.

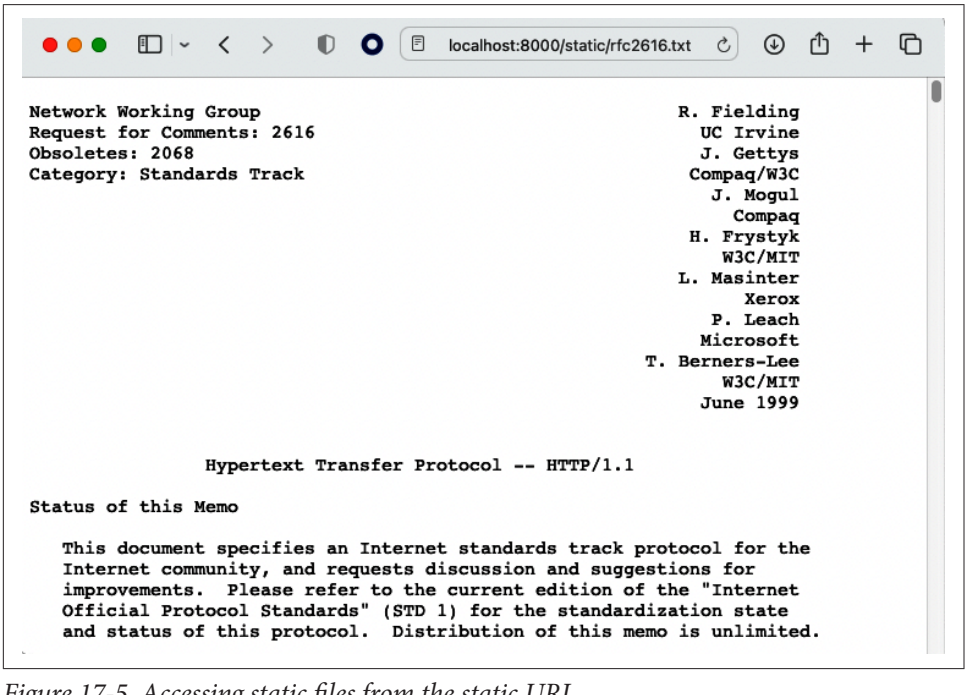

Figure 17-5. Accessing static files from the static URL

You should be aware that `http.FileServer` will list all the contents of the directory that it is serving. This is might not be a major problem if you don't care if the user can see the contents of the directory. Figure 17-6 is a screenshot of a browser showing the contents of the directory.

Figure 17-6. Directory list

However, if you want to prevent this, you can create and place an *index.html* file in the directory. This file will be served instead of the directory listing. *index.html* can be anything, including being empty; it just needs to be there. Figure 17-7 is a screenshot of a browser with *index.html* being shown instead.

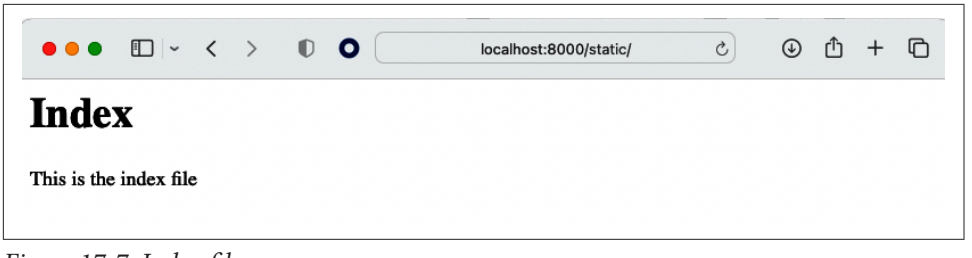

Figure 17-7. Index file

If you want to serve a single file, just use `http.ServeFile`:

```
func main() {
        file := func(w http.ResponseWriter, r *http.Request) {
                http.ServeFile(w, r, "./static/rfc2616.txt")
        }
        http.HandleFunc("/static/http_rfc", file)
        http.ListenAndServe(":8000", nil)
}
```

The `http.ServeFile` function takes in an `http.ResponseWriter`, an `http.Request`, and the path to the file you want to serve. If you wrap it around an anonymous function, you can use it as a handler function, which we did in the example. Then you can use any URL you want to represent the file. Figure 17-8 is a screenshot of a browser showing the file that was served.

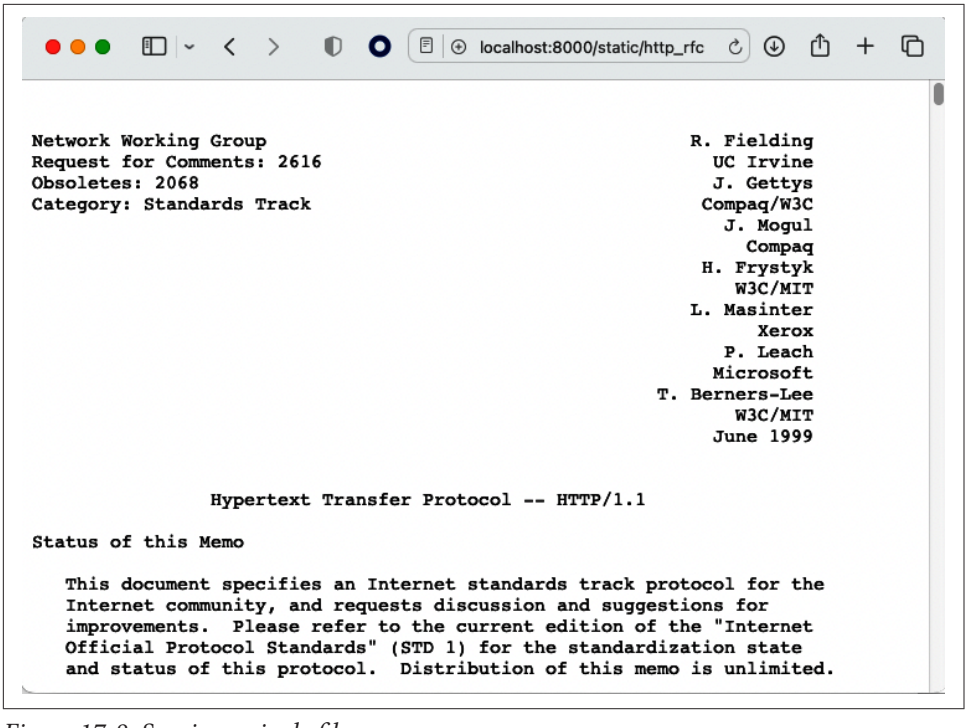

Figure 17-8. Serving a single file

17.6 Creating a JSON Web Service API

Problem

You want to create a simple web service API that returns JSON.

Solution

Use the `net/http` package to create a web service API and the `encoding/json` package to encode data to be sent back as JSON.

Discussion

Web applications are software programs used by humans. Web services are software programs used by other software programs to exchange data between themselves. Go is a popular language for creating web services.

In this recipe, you'll create a web service API that returns a list of people in JSON format. You'll use the `net/http` and `chi` packages to create the web service API and the `encoding/json` package to encode data to be sent back as JSON.

Start by creating a `Person` struct:

```
type Person struct {
        Name      string `json:"name"`
        Height    string `json:"height"`
        Mass      string `json:"mass"`
        HairColor string `json:"hair_color"`
        SkinColor string `json:"skin_color"`
        EyeColor  string `json:"eye_color"`
        BirthYear string `json:"birth_year"`
        Gender    string `json:"gender"`
}

var list []Person

func init() {
        file, _ := os.Open("people.json")
        defer file.Close()
        data, _ := io.ReadAll(file)
        json.Unmarshal(data, &list)
}
```

You use the `init` function to initialize the `list` variable with the data from the *people.json* file. This will be the data that you'll return as JSON:

```
[
    {
    "name": "Luke Skywalker",
    "height": "172",
    "mass": "77",
    "hair_color": "blond",
    "skin_color": "fair",
    "eye_color": "blue",
    "birth_year": "19BBY",
    "gender": "male"
    },
    {
    "name": "C-3PO",
    "height": "167",
    "mass": "75",
    "hair_color": "n/a",
    "skin_color": "gold",
    "eye_color": "yellow",
    "birth_year": "112BBY",
    "gender": "n/a"
    },
    {
    "name": "R2-D2",
    "height": "96",
    "mass": "32",
    "hair_color": "n/a",
    "skin_color": "white, blue",
    "eye_color": "red",
```

```
    "birth_year": "33BBY",
    "gender": "n/a"
    }
]
```

Now you create a handler that uses the pattern `"/people/{id}"` to create a RESTful API that uses the path pattern `/people/<id>`:

```
func main() {
        mux := chi.NewRouter()
        mux.Get("/people/{id}", people)
        http.ListenAndServe(":8000", mux)
}
```

Next is the `people` handler function:

```
func people(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Content-Type", "application/json")
        idstr := chi.URLParam(r, "id")
        id, err := strconv.Atoi(idstr)
        if err != nil {
                w.WriteHeader(http.StatusBadRequest)
                return
        }
        if id < 0 || id >= len(list) {
                w.WriteHeader(http.StatusNotFound)
                return
        }
        json.NewEncoder(w).Encode(list[id])
}
```

First, set the `Content-Type` header to `application/json`. This tells the client that the response is in JSON format. Next, get the `id` from the URL path using the `chi.URLParam` function. If the `id` is not a number, you return a 400 Bad Request error. If the `id` is out of range, you return a 404 Not Found error.

You might be wondering why there isn't a recipe that shows a web service that receives a JSON API. This is because processing JSON was covered in Chapter 9, and handling HTTP requests was covered in Recipe 17.2. You simply take the body of the request and unmarshal it into a struct; then, you can process the rest as you like.

17.7 Serving Through HTTPS

Problem

You want to serve your web application through HTTPS.

Solution

Use the `http.ListenAndServeTLS` function to serve your web application through HTTPS.

Discussion

Most web applications need to be served through HTTPS. This allows the communication between the client and the server to be encrypted. This is important because the communication between the client and the server can contain sensitive information such as passwords and credit card numbers. In some cases, this is mandated; for example, if you accept credit card payments, you need to be PCI-compliant (Payment Card Industry Data Security Standard) and to be PCI-compliant, you need to encrypt the communications between the client and the server.

HTTPS is nothing more than layering HTTP on top of the Transport Security Layer (TLS). The `net/http` package provides the `http.ListenAndServeTLS` function to serve your web application through HTTPS. Go back to the simple web application and see how you can use it to serve your web application through HTTPS:

```
package main

import (
        "net/http"
)

func main() {
        http.HandleFunc("/", index)
        http.ListenAndServeTLS(":8000", "cert.pem", "key.pem", nil)
}

func index(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("Hello World"))
}
```

Most of the code is the same, except you need a certificate file *cert.pem* and a private key file *key.pem*. The *cert.pem* is the SSL certificate, while *key.pem* is the private key for the server. In a production scenario, you will need to get the SSL certificate from a certificate authority (CA) like VeriSign or Thawte or Comodo SSL, or you can use Let's Encrypt to get a free one. However, if you just need a certificate and private key to try things out, you can generate self-signed ones using OpenSSL.

Run this from the command line:

```
$ openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365
-nodes
```

OpenSSL is an open source implementation of SSL and TLS. The library contains a command-line tool with the same name that can do a number of things, including creating private keys and SSL certificates.

The `req` command is used to manage certificate requests but can also be used to create self-signed certificates. Option `-509` tells the tool to create self-signed certificates (X.509 is an International Telecommunication Union standard defining the format of public key certificates). The option `-newkey` tells the tool to generate a new private key. The argument `rsa:4096` tells the tool to create a key that is of size 4,096 bits. The `-keyout` and `-out` options tell the tool to create the private key and certificate files with the respective names given in the argument. The option `-days` specifies the number of days to certify the certificate. Here you use 365, which means the certificate is valid only for 1 year. Finally, use the `-nodes` option (which means, "no DES" rather than "nodes") to say you don't want to encrypt the private key.

When you run the command, you should see something like this:

```
$ openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 3650
-nodes
Generating a 4096 bit RSA private key
.....................................................................
.................................+++
.....................................................................
.....................................................................
.............................................+++
writing new private key to 'key.pem'
\-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
\-----
Country Name (2 letter code) []:SG
State or Province Name (full name) []:
Locality Name (eg, city) []:
Organization Name (eg, company) []:Go Cookbook
Organizational Unit Name (eg, section) []:
Common Name (eg, fully qualified host name) []:localhost
Email Address []:sausheong@gmail.com
$ go run main.go
```

This is the interactive mode. You can press the return key to leave the entries empty except for the common name. The common name is the fully qualified domain name of the server you want to protect. In this case, it's for testing, so you can just enter `localhost`.

Once you're done, you should have two files—the *cert.pem* and *key.pem* files. Copy them into the same directory as the *main.go* file. Then run the code again to start the server.

Now try to access the web application through HTTPS. In a browser, go to *https://localhost:8000*. Your browser might try to warn you that the site is insecure because it is self-signed, after all. You can ignore the warning and proceed to the site. You should see the `Hello World` message as shown in Figure 17-9.

Figure 17-9. HTTPS

If you double-click the lock icon in the address bar, you can see the details of the certificate: the certificate is issued by `Go Cookbook` and is valid for 1 year. You can also see the details of the certificate, including the public key and the signature (see Figure 17-10).

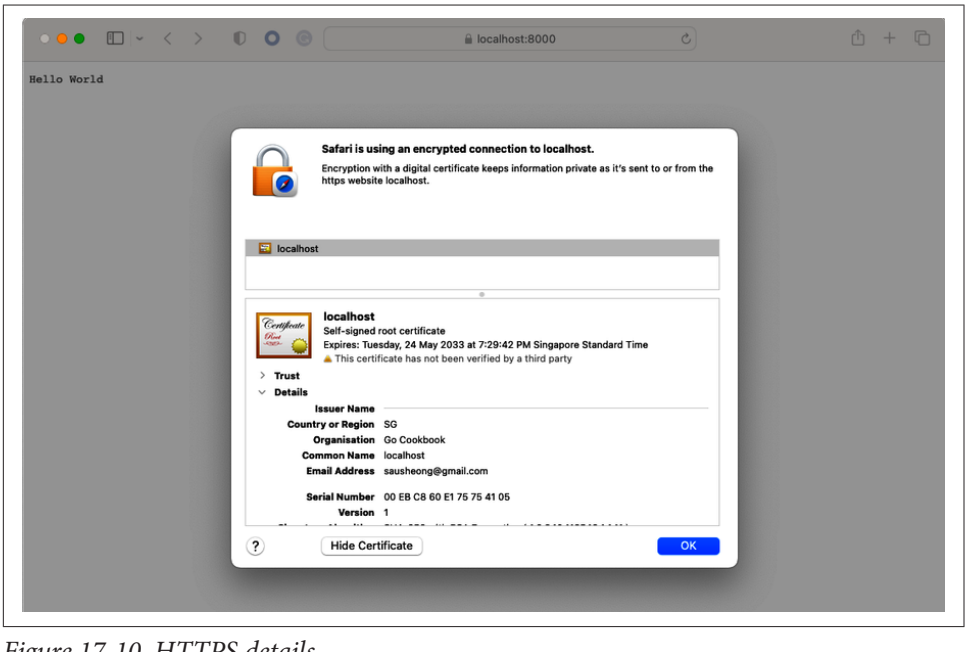

Figure 17-10. HTTPS details

While there would be uses for configuring TLS this way, most likely, if you are deploying your web application in a production environment, you will use a reverse proxy like Nginx or Apache to handle the TLS termination. This way, you can offload the TLS termination, which will improve the performance of your web application. This means your web application can still run with HTTP while it is fronted by a reverse proxy that runs through HTTPS.

17.8 Using Templates for Go Web Applications

Problem

You want to use Go's templating system to create a web application.

Solution

Use the `html/template` package to create a web application.

Discussion

In the earlier web application, you simply wrote to the `http.ResponseWriter` directly. This is fine for simple applications, but it's not very flexible. If you want to create a more complex web application, you'll need to use a template engine.

As a quick recap, the template engine is one of the three parts of a web application, and it is used to render the body of the response. It combines one or more templates with data and returns the response body.

You can use the `html/template` package to create templates.

Take a look at a simple example:

```
package main

import (
        "html/template"
        "net/http"
)

func hello(w http.ResponseWriter, r *http.Request) {
        t, _ := template.ParseFiles("hello.html")
        t.Execute(w, "Hello World!")
}

func main() {
        http.HandleFunc("/", hello)
        http.ListenAndServe(":8000", nil)
}
```

The only difference from the earlier web application is that you're using the `html/template` package to parse the template file into a *template*, which is an instance of `template.Template`. You then call the `Execute` method on the template, passing it the data to be rendered.

In the template file *hello.html*, you can use the `{{.}}` syntax to render the data:

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Go Cookbook</title>
  </head>
  <body>
    <h1>{{ . }}</h1>
  </body>
</html>
```

The dot (.) between the double braces is an *action*, and it's a command for the Go template engine to replace it with a value when the template is executed. In this case, the value is the string "Hello World!"

If you run the program and open the browser, you'll see the output shown in Figure 17-11.

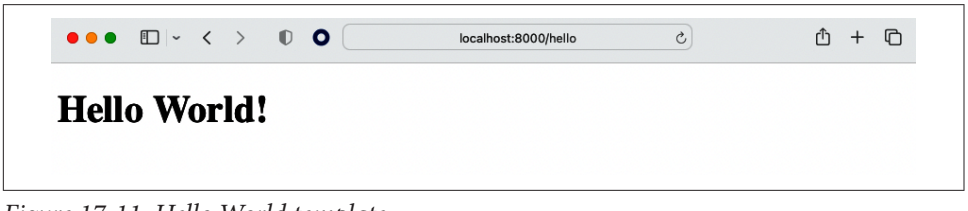

Figure 17-11. Hello World template

Actions are a critical part of the template engine. They are used to control the flow of the template, and they can also be used to perform operations on the data in the template.

Some of the more commonly used actions are:

- The most important commonly used action is the dot `{{.}}` action. It's used to render the data passed to the template.
- The `{{range}}` action is used to iterate over a slice or a map.
- The `{{if}}`, `{{else}}` action is used to perform conditional checks.

The dot action represents the data passed to the template. It can be a string or an integer, any primitive type, or it can be a struct, slice, or map. With a struct, you can access the fields using the dot action.

Take this struct:

```
type Person struct {
        Name      string
        Gender    string
        Homeworld string
}
```

Send this struct to the template from the handler:

```
func person(w http.ResponseWriter, r *http.Request) {
        t, _ := template.ParseFiles("person.html")
        t.Execute(w, Person{
                Name:      "Luke Skywalker",
                Gender:    "male",
                Homeworld: "Tatooine",
        })
}
```

In the template file, you can access the fields using the dot action:

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Go Cookbook</title>
  </head>
  <body>
    <h1>{{ .Name }}</h1>
    <ul>
      <li><b>Gender:</b> {{ .Gender }}</li>
      <li><b>Home world:</b> {{ .Homeworld }}</li>
    </ul>
  </body>
</html>
```

If you run the program and open the browser, you'll see the output shown in Figure 17-12.

Figure 17-12. Person template

For actions that wrap around a section of the template, the opening action is followed by a {{end}} action. For example, the {{range}} action is followed by a {{end}} action.

Within these wrapped-around sections, you can use the dot action to render the data. For example, in the {{range}} action, you can use the dot action to render the data in the slice or map.

Here's how this works:

```
func people(w http.ResponseWriter, r *http.Request) {
        t, _ := template.ParseFiles("people.html")
        t.Execute(w, []string{"Luke", "Leia", "Han", "Chewbacca"})
}


func main() {
        http.HandleFunc("/people", people)
        http.ListenAndServe(":8000", nil)
}
```

As you can see, you are passing to the *people.html* template a slice of names. In the template file, you can use the {{range}} action to iterate over the slice and render the data:

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Go Cookbook</title>
  </head>
  <body>
    <div>This is a list of people in Star Wars:</div>
    <ul>
        {{ range . }}
        <li>{{ . }}</li>
        {{ end }}
    </ul>
  </body>
</html>
```

You might have noticed that you are ranging through the dot (.), but within the {{range}} action, you are using the dot action again. The dot action within the range refers to the item in the slice. This is equivalent to:

```
for _, item := range people // people is the dot for the range
        item                // item is the dot within the range
}
```

Figure 17-13 is a screenshot of a browser showing the output of the people template.

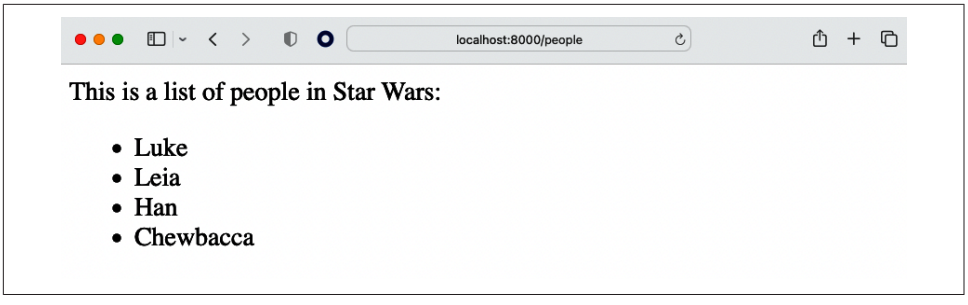

Figure 17-13. People template

The {{if}}, {{else}} action is used to perform conditional checks. For example, you can use it to check if the slice is empty.

Send an empty slice to the *people.html* template:

```
func people(w http.ResponseWriter, r *http.Request) {
        t, _ := template.ParseFiles("people.html")
        t.Execute(w, []string{})
}
```

Now you can use the {{if}}, {{else}} action to check if the slice is empty:

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Go Cookbook</title>
  </head>
  <body>
    {{ if gt (len .) 0 }}
    <div>This is a list of people in Star Wars:</div>
    <ul>
        {{ range . }}
        <li>{{ . }}</li>
        {{ end }}
    </ul>
    {{ else }}
    <div>There are no people in this list.</div>
    {{ end }}
  </body>
</html>
```

Take a look at the output again. Figure 17-14 is a screenshot of a browser showing the output of the people template with conditionals.

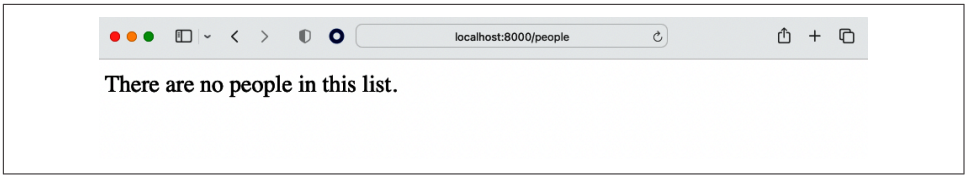

Figure 17-14. No people template

So far, you've been ignoring the error that's returned along with the parsed template. The usual Go practice is to handle the error, but Go provides another mechanism to handle errors returned by parsing templates:

```
t  := template.Must(template.ParseFiles("people.html"))
```

The `Must` function wraps around a function that returns a pointer to a template and an error and panics if the error is not a `nil`. This convenience function pattern is seen elsewhere in the Go standard library and mentioned in Recipe 3.2.

17.9 Making an HTTP Client Request

Problem

You want to make an HTTP request to a web server.

Solution

Use the `net/http` package to make an HTTP request.

Discussion

HTTP is a request-respond protocol, and serving requests is only half of the story. The other half is making requests. The `net/http` package provides functions to make HTTP requests. You will start with the two most common HTTP request methods, `GET` and `POST`, which have their own convenience functions.

The `http.Get` function is the most basic HTTP client function in the `net/http` package. It simply makes a `GET` request to the specified URL and returns an `http.Response` and an error. The `http.Response` has a `Body` field that you can read to get the response body.

Here's a simple example:

```
func main() {
        resp, err := http.Get("https://www.ietf.org/rfc/rfc2616.txt")
        if err != nil {
                // resolve error
        }
```

```
        defer resp.Body.Close()
        body, err := io.ReadAll(resp.Body)
        if err != nil {
                // resolve error
        }
        fmt.Println(string(body))
}
```

The body is a slice of bytes, which you can convert into a string. If you run it on a terminal, you'll get the entire RFC 2616 document in text.

Next, you can make a `POST` request. Start with making a `POST` request that sends a JSON message to a server. For this example, you'll use the *http://httpbin.org/post* endpoint, which will echo back the request body. HTTPBin is an open source tool that provides a set of endpoints you can use to test your HTTP clients.

This example and the following examples will not handle errors for brevity, but you should always handle errors properly in your code:

```
func main() {
        msg := strings.NewReader(`{"message": "Hello, World!"}`)
        resp, _ := http.Post("https://httpbin.org/post", "application/json", msg)
        defer resp.Body.Close()
        body, _ := io.ReadAll(resp.Body)
        fmt.Println(string(body))
}
```

The `http.Post` function takes the URL, the content type, and a request body that is of type `io.Reader` as parameters. You used `strings.NewReader` to create a `string.Reader` (which is an `io.Reader`, of course) from the JSON message, and then pass it to `http.Post` as the request body.

As before, the `http.Post` function returns an `http.Response` and an error, and you can read the response body and print it out.

When you run it from the terminal, you get the following output:

```
% go run main.go
{
  "args": {},
  "data": "{\"message\": \"Hello, World!\"}",
  "files": {},
  "form": {},
  "headers": {
    "Accept-Encoding": "gzip",
    "Content-Length": "28",
    "Content-Type": "application/json",
    "Host": "httpbin.org",
    "User-Agent": "Go-http-client/2.0",
    "X-Amzn-Trace-Id": "Root=1-6342bd52-6167b9e86bafdf0a41bf106a"
  },
  "json": {
```

```
    "message": "Hello, World!"
  },
  "origin": "123.80.47.37",
  "url": "https://httpbin.org/post"
}
```

A very common use case is to send data from an HTML form to a server with the `POST` method. The `http.PostForm` function makes it easy to send form data to a server. It takes the URL and a `url.Values` as parameters.

The `url.Values` type is a map of string keys and string values found in the `net/url` package. The `url.Values` type has an `Add` method that you can use to add values to the map. Each key can have multiple values. The `http.PostForm` function will encode the form data and send it to the server:

```
func main() {
        form := url.Values{}
        form.Add("message", "Hello, World!")
        resp, _ := http.PostForm("https://httpbin.org/post", form)
        defer resp.Body.Close()
        body, _ := io.ReadAll(resp.Body)
        fmt.Println(string(body))
}
```

When you run it from the terminal, you get the following output:

```
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "message": "Hello, World!"
  },
  "headers": {
    "Accept-Encoding": "gzip",
    "Content-Length": "25",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "User-Agent": "Go-http-client/2.0",
    "X-Amzn-Trace-Id": "Root=1-6342bea1-0a79dc8766f32b8f0f855391"
  },
  "json": null,
  "origin": "123.80.47.37",
  "url": "https://httpbin.org/post"
}
```

So far, you have been using convenience methods to make HTTP requests.

The `net/http` package also provides a more generic function to make HTTP requests using `http.Client`, which is quite straightforward. You create an instance of the `http.Client`, and then you call the `Do` method, passing in a `http.Request` as the parameter. This will return an `http.Response` and an error as before.

To make it even simpler, the `net/http` package even provides an `http.DefaultClient` instance that you can use immediately, with default settings.

Here's how this can be done. This time around, you'll add a cookie to the request:

```
func main() {
        req, _ := http.NewRequest("GET", "https://httpbin.org/cookies", nil)
        req.AddCookie(&http.Cookie{
                Name:  "foo",
                Value: "bar",
        })
        resp, _ := http.DefaultClient.Do(req)
        defer resp.Body.Close()
        body, _ := io.ReadAll(resp.Body)
        fmt.Println(string(body))
}
```

First, create an `http.Request` using the `http.NewRequest` function. Then pass in the HTTP method, the URL, and an `io.Reader` as the request body. The `http.NewRequest` function returns an `http.Request` and an error (which you disregard for brevity as in the previous code). Then add a cookie to the request using the `http.Request.AddCookie` method.

Finally, call the `http.DefaultClient.Do` method, passing in the `http.Request` as the parameter. This will return an `http.Response` and an error.

When you run it from the terminal, you get the following output:

```
{
  "cookies": {
    "foo": "bar"
  }
}
```

You can do the same with `POST` requests as well:

```
func main() {
        msg := strings.NewReader(`{"message": "Hello, World!"}`)
        req, _ := http.NewRequest("POST", "https://httpbin.org/post", msg)
        req.Header.Add("Content-Type", "application/json")
        resp, _ := http.DefaultClient.Do(req)
        defer resp.Body.Close()
        body, _ := io.ReadAll(resp.Body)
        fmt.Println(string(body))
}
```

The output is the same as using the `http.Post` function.

So far, you've only been sending `GET` and `POST` requests. The `net/http` package also supports other HTTP methods like `PUT`, `PATCH`, `DELETE`, and so on, using the `http.Client` mechanism.

You can give PUT a try:

```
func main() {
        msg := strings.NewReader(`{"message": "Hello, World!"}`)
        req, _ := http.NewRequest("PUT", "https://httpbin.org/put", msg)
        req.Header.Add("Content-Type", "application/json")
        resp, _ := http.DefaultClient.Do(req)
        defer resp.Body.Close()
        body, _ := io.ReadAll(resp.Body)
        fmt.Println(string(body))
}
```

If you look at the output at the terminal, it's almost the same as with POST:

```
{
  "args": {},
  "data": "{\"message\": \"Hello, World!\"}",
  "files": {},
  "form": {},
  "headers": {
    "Accept-Encoding": "gzip",
    "Content-Length": "28",
    "Content-Type": "application/json",
    "Host": "httpbin.org",
    "User-Agent": "Go-http-client/2.0",
    "X-Amzn-Trace-Id": "Root=1-6342c1d0-0e22519a00dfa15d41ddeee2"
  },
  "json": {
    "message": "Hello, World!"
  },
  "origin": "123.80.47.37",
  "url": "https://httpbin.org/put"
}
```

CHAPTER 18
Testing Recipes

18.0 Introduction

Software testing is the process of checking that the software does what it is supposed to do. It's a critical part of software development. Software testing, like many types of testing activities in other fields, traditionally happens after development completes. It is mostly done by people (testers) who run through scenarios, called *test cases*, and verify the output with the expected results.

Testing happens at various stages of the software development lifecycle and even beyond that. At the lowest level of code, *unit testing* checks code in individual functions and software modules. *Integration testing* ensures that different modules work well together, and *functional testing* ensures the correctness of the output.

Unlike many other fields, software testing doesn't necessarily need to be done after the programs have been written and neither does it always need to be done by humans. Software testing can and often is done through *automated testing* by writing test scripts that execute test cases.

In the *test-driven development* (TDD) methodology, automated test cases are written before any code is written and are repeatedly executed as the code is being written until the test cases succeed. In *continuous testing*, automated testing is done continuously throughout the software development lifecycle.

As you have realized, automated testing is pretty important in software development. In Go, testing is built into the language itself, and Go provides the `go test` tool and the `testing` package to automate it.

18.1 Automating Functional Tests

Problem

You want to automate functional testing of a function.

Solution

Create a test function and use the `go test` tool to run it.

Discussion

Go provides a minimalist set of built-in tools for testing with the `go test` command-line tool and the `testing` package. These tools are used for both functional and performance testing. In this recipe, we'll focus on functional testing.

Go back to our trusty `Add` function and see how you can test it. You place the function in a package called `arith`:

```
package arith

func Add(a, b int) int {
        return a + b
}
```

First, create a file named *testing_test.go*. The `go test` tool will look for all files in the same package that end with *_test.go* and consider those files to contain test functions:

```
package arith

import "testing"

func TestAdd(t *testing.T) {
        result := Add(1, 2)
        if result != 3 {
                t.Error("Adding 1 and 2 doesn't produce 3")
        }
}
```

The name of the package the file is in must be the same as the one with the function you want to test.

Each test function starts with "Test," followed by "<*Xxx*>," and you use camel case to describe what you're testing. There must be only a single input parameter to the test function, which is a pointer to `testing.T`. `T` is a struct instance that is passed to test functions to manage the test state and also for logging test results.

The most straightforward use of `T` is how it's used in `TestAdd` where you called the `Error` function to signal failure in the test. You can also call various similar functions like `Errorf`, `Fail`, `Fatal`, and `Fatalf`. For logging test results, you can use the functions `Log` and `Logf`:

```
func TestAdd(t *testing.T) {
        result := Add(1, 2)
        if result != 3 {
                t.Error("Adding 1 and 2 doesn't produce 3")
        } else {
                t.Log("Result is:", result)
        }
}
```

To run the test, you use the `go test` tool at the command line, using the `-v` (verbose) flag to show more information:

```
% go test -v
=== RUN   TestAdd
    testing_test.go:10: Result is: 3
--- PASS: TestAdd (0.00s)
PASS
ok      github.com/sausheong/gocookbook/ch18_testing    0.446s
```

Let's say you want to skip the test for now; you can call `SkipNow` to skip the test function:

```
func TestAdd(t *testing.T) {
        t.SkipNow()
        result := Add(1, 2)
        if result != 3 {
                t.Error("Adding 1 and 2 doesn't produce 3")
        } else {
                t.Log("Result is:", result)
        }
}
```

When you run the test, this is what you'll get:

```
% go test -v
=== RUN   TestAdd
--- SKIP: TestAdd (0.00s)
PASS
ok      github.com/sausheong/gocookbook/ch18_testing    0.530s
```

18.2 Running Multiple Test Cases

Problem

You want to run multiple test cases without having to set up different test functions.

Solution

Run table-driven tests by providing a set of test cases.

Discussion

You wouldn't normally want to test a function with just one set of data. Usually, you would want to provide a set of test data that covers multiple scenarios and boundary cases.

Table-driven testing is a simple technique that provides a set of test cases consisting of inputs and expected results all in one go and then iterates through them in a single test function:

```
func TestAddWithTables(t *testing.T) {
        testCases := []struct {
                a      int
                b      int
                result int
        }{
                {1, 2, 3},
                {12, 30, 42},
                {100, -1, 99},
        }

        for _, testCase := range testCases {
                result := Add(testCase.a, testCase.b)
                if result != testCase.result {
                        t.Errorf("Adding %d and %d doesn't produce %d, instead \
                                it produces %d",
                                testCase.a, testCase.b, testCase.result, result)
                }
        }
}
```

If you run the test now, it will go through all the test cases and passes only when all of them pass. Let's see what happens if you change the `Add` function to return `a * b` instead of `a + b`:

```
func Add(a, b int) int {
        return a * b
}
```

When you run the test, you see how each test case fails:

```
% go test -v -run TestAdd
=== RUN   TestAddWithTables
    testing_test.go:30: Adding 1 and 2 doesn't produce 3, instead it produces 2
    testing_test.go:30: Adding 12 and 30 doesn't produce 42, instead it produces
        360
    testing_test.go:30: Adding 100 and -1 doesn't produce 99, instead it produces
        -100
```

```
--- FAIL: TestAddWithTables (0.00s)
FAIL
exit status 1
FAIL    github.com/sausheong/gocookbook/ch18_testing    0.423s
```

18.3 Setting Up and Tearing Down Before and After Tests

Problem

You want to set up data and an environment for testing and tear it down after the test is run.

Solution

You can create helper functions or use the `TestMain` feature to control the flow of the test functions.

Discussion

Testing often needs data and an environment set up that is used in testing. These are called *test fixtures*. While test fixtures are usually destroyed outside of the scope of the test function, sometimes certain artifacts like files, database records, or configuration files need to be removed so they won't interfere with subsequent testing.

There are a couple of ways to set up and tear down test fixtures.

The easiest way is to create helper functions. For example, you want to take an image file and flip it. In this example, you want to flip a PNG-format image of the Mona Lisa.

The algorithm is easy; first load the image file into a two-dimensional grid of pixels (where a pixel is represented by a `color.Color` struct). (For more information about the algorithm, see Recipe 15.4.)

The specific code you want to test is the `flip` function that takes the grid and flips the pixels in it:

```
func flip(grid [][]color.Color) {
        for x := 0; x < len(grid); x++ {
                col := grid[x]
                for y := 0; y < len(col)/2; y++ {
                        k := len(col) - y - 1
                        col[y], col[k] = col[k], col[y]
                }
        }
}
```

The obvious setup is to take a PNG file and load it into a grid to get it ready for testing.

After flipping the grid, you'll save it to a file, then load it up again to check the dimensions. This leaves a file on the filesystem after the test is run, so you need to clean that up.

The straightforward way to do this is to call `setup` at the beginning of the test function and also call `teardown` before the end of the test function. Another way of doing this is to create a `setup` function that returns a teardown closure:

```
func setup(filename string) (teardown func(tempfile string),
grid [][]color.Color) {
        grid = load(filename)
        teardown = func(tempfile string) {
                os.Remove(tempfile)
        }
        return
}
```

You can then call the teardown using `defer`, which will ensure that the teardown happens at the end of the function:

```
func TestFlip(t *testing.T) {
        teardown, grid := setup("monalisa.png")
        defer teardown("flipped.png")
        flip(grid)
        save("flipped.png", grid)
        g := load("flipped.png")
        if len(g) != 321 || len(g[0]) != 480 {
                t.Error("Grid is wrong size", "width:", len(g), "length:",
                        len(g[0]))
        }
}
```

A *test suite* is a collection of test cases. Helper functions work well for smaller test suites, but in a larger one, it is tedious to call the helper function over and over again, especially if it takes up resources. Normally you would want to set up all the test fixtures up front and then tear it all down after the test suite completes.

The `TestMain` feature was added in Go 1.4, and it allows more flexible and lower-level control of setting up and tearing down test fixtures. If a test file contains the `TestMain` function, then Go will run `TestMain` instead of the tests directly. `TestMain` runs in the main goroutine and will run the rest of the test cases when you call `m.Run`. As a result, you can set up whatever you need before `m.Run` and tear down whatever fixtures were created after.

When you call `m.Run`, it will return an exit code that you can pass to `os.Exit` to exit the test suite cleanly:

```
func TestMain(m *testing.M) {
        fmt.Println("setup")
        exitCode := m.Run()
        fmt.Println("teardown")
        os.Exit(exitCode)
}
```

When you run `go test` in the command line, you will get this:

```
% go test -v
setup
=== RUN   TestAdd
    testing_test.go:23: Result is: 3
--- PASS: TestAdd (0.00s)
=== RUN   TestAddWithTables
--- PASS: TestAddWithTables (0.00s)
=== RUN   TestFlip
--- PASS: TestFlip (0.07s)
PASS
teardown
ok      github.com/sausheong/gocookbook/ch18_testing    0.527s
```

As you can see, the setup runs before any test functions are run, and the teardown runs after all the test functions complete.

18.4 Creating Subtests to Have Finer Control Over Groups of Test Cases

Problem

You want to create subtests within a test function to have finer control over test cases.

Solution

Use the `t.Run` function to create subtests within a test function. Subtests extend the flexibility of test functions to another level down.

Discussion

When using table-driven tests, you often want to run specific tests or have finer-grained control over the test cases. However, table-driven tests are really driven through data so there is no way you can control it. For example, in the previous test function using table-driven tests you can only test the results for nonequality for every single test case:

```
func TestAddWithTables(t *testing.T) {
        testCases := []struct {
                a      int
                b      int
                result int
        }{
                {1, 2, 3},
                {12, 30, 42},
                {100, -1, 99},
        }

        for _, testCase := range testCases {
                result := Add(testCase.a, testCase.b)
                if result != testCase.result {
                        t.Errorf("Adding %d and %d doesn't produce %d, instead \
                                it produces %d",
                                testCase.a, testCase.b, testCase.result, result)
                }
        }
}
```

Go 1.7 added a new feature to allow subtests within a test function, using the `t.Run` function. This is how you can turn your table-driven test function into one that uses subtests:

```
func TestAddWithSubTest(t *testing.T) {
        testCases := []struct {
                name   string
                a      int
                b      int
                result int
        }{
                {"OneDigit", 1, 2, 3},
                {"TwoDigits", 12, 30, 42},
                {"ThreeDigits", 100, -1, 99},
        }
        for _, testCase := range testCases {
                t.Run(testCase.name, func(t *testing.T) {
                        result := Add(testCase.a, testCase.b)
                        if result != testCase.result {
                                t.Errorf("Adding %d and %d doesn't produce %d, \
                                        instead it produces %d",
                                        testCase.a, testCase.b, testCase.result,
                                        result)
                        }
                })
        }
}
```

As you can see, you added a test name to each of the test cases. Then within the `for` loop, you call `t.Run`, passing it the test case name and also calling an anonymous

function that has the same form as a normal test function. This is the subtest from which you can run each test case.

Run the test now and see what happens:

```
% go test -v -run TestAddWithSubTest
=== RUN   TestAddWithSubTest
=== RUN   TestAddWithSubTest/OneDigit
=== RUN   TestAddWithSubTest/TwoDigits
=== RUN   TestAddWithSubTest/ThreeDigits
--- PASS: TestAddWithSubTest (0.00s)
    --- PASS: TestAddWithSubTest/OneDigit (0.00s)
    --- PASS: TestAddWithSubTest/TwoDigits (0.00s)
    --- PASS: TestAddWithSubTest/ThreeDigits (0.00s)
PASS
ok      github.com/sausheong/gocookbook/ch18_testing    0.607s
```

As you can see, each subtest is named and run separately under the umbrella test function. You could, in fact, pick and choose the subtest you want to run:

```
% go test -v -run TestAddWithSubTest/TwoDigits
=== RUN   TestAddWithSubTest
=== RUN   TestAddWithSubTest/TwoDigits
--- PASS: TestAddWithSubTest (0.00s)
    --- PASS: TestAddWithSubTest/TwoDigits (0.00s)
PASS
ok      github.com/sausheong/gocookbook/ch18_testing    0.193s
```

In the preceding example, you didn't do anything more than what you did before, but you could have done more to customize the setup and teardown. For example:

```
func TestAddWithCustomSubTest(t *testing.T) {
        testCases := []struct {
                name     string
                a        int
                b        int
                result   int
                setup    func()
                teardown func()
        }{
                {"OneDigit", 1, 2, 3,
                        func() { fmt.Println("setup one") },
                        func() { fmt.Println("teardown one") }},
                {"TwoDigits", 12, 30, 42,
                        func() { fmt.Println("setup two") },
                        func() { fmt.Println("teardown two") }},
                {"ThreeDigits", 100, -1, 99,
                        func() { fmt.Println("setup three") },
                        func() { fmt.Println("teardown three") }},
        }
        for _, testCase := range testCases {
                t.Run(testCase.name, func(t *testing.T) {
                        testCase.setup()
```

```
                        defer testCase.teardown()
                        result := Add(testCase.a, testCase.b)
                        if result != testCase.result {
                                t.Errorf("Adding %d and %d doesn't produce %d, \
                                        instead it produces %d",
                                        testCase.a, testCase.b, testCase.result,
                                        result)
                        } else {
                                fmt.Println(testCase.name, "ok.")
                        }
                })
        }
}
```

If you run it now, you can see that each subtest has its own setup and teardown functions that are called separately:

```
% go test -v -run TestAddWithCustomSubTest
=== RUN   TestAddWithCustomSubTest
=== RUN   TestAddWithCustomSubTest/OneDigit
setup one
OneDigit ok.
teardown one
=== RUN   TestAddWithCustomSubTest/TwoDigits
setup two
TwoDigits ok.
teardown two
=== RUN   TestAddWithCustomSubTest/ThreeDigits
setup three
ThreeDigits ok.
teardown three
--- PASS: TestAddWithCustomSubTest (0.00s)
    --- PASS: TestAddWithCustomSubTest/OneDigit (0.00s)
    --- PASS: TestAddWithCustomSubTest/TwoDigits (0.00s)
    --- PASS: TestAddWithCustomSubTest/ThreeDigits (0.00s)
PASS
ok      github.com/sausheong/gocookbook/ch18_testing    0.278s
```

In the previous examples, you used subtests on a table-driven test. However, it doesn't need to be table-driven. Subtests can be used simply to group different tests under a single test function. For example, in the following case, you want to group the tests on the `flip` function under a single test function:

```
func TestFlipWithSubTest(t *testing.T) {
        grid := load("monalisa.png") // setup for all flip tests
        t.Run("CheckPixels", func(t *testing.T) {
                p1 := grid[0][0]
                flip(grid)
                defer flip(grid) // teardown for check pixel to unflip the grid
                p2 := grid[0][479]
                if p1 != p2 {
                        t.Fatal("Pixels not flipped")
                }
```

```
        })

        t.Run("CheckDimensions", func(t *testing.T) {
                flip(grid)
                save("flipped.png", grid)
                // teardown for check dimensions to remove the file
                defer os.Remove("flipped.png")
                g := load("flipped.png")
                if len(g) != 321 || len(g[0]) != 480 {
                        t.Error("Grid is wrong size", "width:", len(g),
                        "length:", len(g[0]))
                }
        })
}
```

In this case, you have a single setup for all the `flip` tests but different teardowns for individual test cases. Each test case can be run as a different test function, but grouping them together allows you to have a one-time setup for test fixtures and also to run multiple subtests under a single umbrella:

```
% go test -v -run TestFlipWithSubTest
=== RUN   TestFlipWithSubTest
=== RUN   TestFlipWithSubTest/CheckPixels
=== RUN   TestFlipWithSubTest/CheckDimensions
--- PASS: TestFlipWithSubTest (0.07s)
    --- PASS: TestFlipWithSubTest/CheckPixels (0.00s)
    --- PASS: TestFlipWithSubTest/CheckDimensions (0.05s)
PASS
ok      github.com/sausheong/gocookbook/ch18_testing    0.269s
```

18.5 Running Tests in Parallel

Problem

You want to speed up testing by running tests in parallel.

Solution

Use the `t.Parallel` function to enable tests or subtests to run in parallel.

Discussion

By default, test functions in the same package are run sequentially. Go 1.7 included a new function, `t.Parallel`, that allows test functions to be run in parallel. Doing this is very straightforward. You only need to add a line that calls `t.Parallel` in your test functions. Here is a quick look at some simple test functions:

```
func TestAddOneDigit(t *testing.T) {
        result := Add(1, 2)
        if result != 3 {
                t.Error("Adding 1 and 2 doesn't produce 3")
        }
}

func TestAddTwoDigits(t *testing.T) {
        result := Add(12, 30)
        if result != 42 {
                t.Error("Adding 12 and 30 doesn't produce 42")
        }
}

func TestAddThreeDigits(t *testing.T) {
        result := Add(100, -1)
        if result != 99 {
                t.Error("Adding 100 and -1 doesn't produce 99")
        }
}
```

You also add a 0.5-second delay in the `Add` function to highlight the test timing:

```
func Add(a, b int) int {
        time.Sleep(500 * time.Millisecond)
        return a + b
}
```

If you run the tests, you can see that the tests are run in sequence. Including other overheads, the overall test timing is almost 2 seconds:

```
% go test -v
=== RUN   TestAddOneDigit
--- PASS: TestAddOneDigit (0.50s)
=== RUN   TestAddTwoDigits
--- PASS: TestAddTwoDigits (0.50s)
=== RUN   TestAddThreeDigits
--- PASS: TestAddThreeDigits (0.50s)
PASS
ok      github.com/sausheong/gocookbook/ch18_testing    1.997s
```

By adding a single line in each of the test functions, you can run the code in parallel:

```
func TestAddOneDigit(t *testing.T) {
        t.Parallel()
        result := Add(1, 2)
        if result != 3 {
                t.Error("Adding 1 and 2 doesn't produce 3")
        }
}

func TestAddTwoDigits(t *testing.T) {
        t.Parallel()
        result := Add(12, 30)
```

```
        if result != 42 {
                t.Error("Adding 12 and 30 doesn't produce 42")
        }
}

func TestAddThreeDigits(t *testing.T) {
        t.Parallel()
        result := Add(100, -1)
        if result != 99 {
                t.Error("Adding 100 and -1 doesn't produce 99")
        }
}
```

Now run it again:

```
% go test -v
=== RUN   TestAddOneDigit
=== PAUSE TestAddOneDigit
=== RUN   TestAddTwoDigits
=== PAUSE TestAddTwoDigits
=== RUN   TestAddThreeDigits
=== PAUSE TestAddThreeDigits
=== CONT  TestAddOneDigit
=== CONT  TestAddTwoDigits
=== CONT  TestAddThreeDigits
--- PASS: TestAddTwoDigits (0.50s)
--- PASS: TestAddOneDigit (0.50s)
--- PASS: TestAddThreeDigits (0.50s)
PASS
ok      github.com/sausheong/gocookbook/ch18_testing    0.686s
```

You can see how the test functions now run in parallel, and the timing is reduced to almost 0.5 seconds.

You can also run subtests in parallel, but you need to be careful because there's a big gotcha here. Let's take whatever you've done in the test functions and translate that into subtests, making it run in parallel:

```
func TestAddWithSubTestAndParallel(t *testing.T) {
        testCases := []struct {
                name   string
                a      int
                b      int
                result int
        }{
                {"OneDigit", 1, 2, 3},
                {"TwoDigits", 12, 30, 42},
                {"ThreeDigits", 100, -1, 99},
        }
        for _, testCase := range testCases {
                t.Run(testCase.name, func(t *testing.T) {
                        t.Parallel()
                        result := Add(testCase.a, testCase.b)
```

```
                        if result != testCase.result {
                                t.Errorf("Adding %d and %d doesn't produce %d,
                                        instead it produces %d",
                                        testCase.a, testCase.b, testCase.result,
                                        result)
                        }
                })
        }
}
```

When you run it, it looks fine:

```
% go test -v -run TestAddWithSubTestAndParallel
=== RUN   TestAddWithSubTestAndParallel
=== RUN   TestAddWithSubTestAndParallel/OneDigit
=== PAUSE TestAddWithSubTestAndParallel/OneDigit
=== RUN   TestAddWithSubTestAndParallel/TwoDigits
=== PAUSE TestAddWithSubTestAndParallel/TwoDigits
=== RUN   TestAddWithSubTestAndParallel/ThreeDigits
=== PAUSE TestAddWithSubTestAndParallel/ThreeDigits
=== CONT  TestAddWithSubTestAndParallel/OneDigit
=== CONT  TestAddWithSubTestAndParallel/TwoDigits
=== CONT  TestAddWithSubTestAndParallel/ThreeDigits
--- PASS: TestAddWithSubTestAndParallel (0.00s)
    --- PASS: TestAddWithSubTestAndParallel/TwoDigits (0.50s)
    --- PASS: TestAddWithSubTestAndParallel/OneDigit (0.50s)
    --- PASS: TestAddWithSubTestAndParallel/ThreeDigits (0.50s)
PASS
ok      github.com/sausheong/gocookbook/ch18_testing    0.718s
```

This looks fine because you know if it doesn't run in parallel, it would take almost 2 seconds. But is it really OK? Let's check by adding a single line to check the actual test case that was run in each subtest:

```
...
t.Parallel()
t.Logf("Test case %s with inputs %d and %d should produce %d",
    testCase.name, testCase.a, testCase.b, testCase.result)
result := Add(testCase.a, testCase.b)
...
```

Run it again and see the results:

```
% go test -v -run TestAddWithSubTestAndParallel
=== RUN   TestAddWithSubTestAndParallel
=== RUN   TestAddWithSubTestAndParallel/OneDigit
=== PAUSE TestAddWithSubTestAndParallel/OneDigit
=== RUN   TestAddWithSubTestAndParallel/TwoDigits
=== PAUSE TestAddWithSubTestAndParallel/TwoDigits
=== RUN   TestAddWithSubTestAndParallel/ThreeDigits
=== PAUSE TestAddWithSubTestAndParallel/ThreeDigits
=== CONT  TestAddWithSubTestAndParallel/OneDigit
=== CONT  TestAddWithSubTestAndParallel/ThreeDigits
=== CONT  TestAddWithSubTestAndParallel/TwoDigits
```

```
=== CONT  TestAddWithSubTestAndParallel/OneDigit
    testing_test.go:108: Test case ThreeDigits with inputs 100 and -1 should
        produce 99
=== CONT  TestAddWithSubTestAndParallel/TwoDigits
    testing_test.go:108: Test case ThreeDigits with inputs 100 and -1 should
        produce 99
=== CONT  TestAddWithSubTestAndParallel/ThreeDigits
    testing_test.go:108: Test case ThreeDigits with inputs 100 and -1 should
        produce 99
--- PASS: TestAddWithSubTestAndParallel (0.00s)
    --- PASS: TestAddWithSubTestAndParallel/ThreeDigits (0.50s)
    --- PASS: TestAddWithSubTestAndParallel/TwoDigits (0.50s)
    --- PASS: TestAddWithSubTestAndParallel/OneDigit (0.50s)
PASS
ok      github.com/sausheong/gocookbook/ch18_testing    0.691s
```

The last test case was run three times in parallel! What happened? This happened because you are trying to use a goroutine (by calling `t.Run`) on a loop iterator variable (the `testCase` variable). The second parameter in `t.Run` is a closure that is bound to the same `testCase` variable in every iteration, and a pointer to this variable is passed into the closure. The iterator and the goroutines run independently, and the iterator (in this case) ran and finished faster than the goroutine could even start, so the `testCase` variable ends up being assigned the last test case.

This is a well-known problem. Normally, what you should do is pass the variable into the closure by value instead of using it directly from the iterator. However, this is not possible here because `t.Run` expects a function with only one parameter, which is `testing.T`. So how can you avoid this?

The simplest fix is to make `testCase` a local variable within the loop instead. This is because variables declared within the loop are not shared between iterations:

```
for _, tc := range testCases {
    testCase := tc
    t.Run(testCase.name, func(t *testing.T) {
        t.Parallel()
        t.Logf("Test case %s with inputs %d and %d should produce %d",
            testCase.name, testCase.a, testCase.b, testCase.result)
        result := Add(testCase.a, testCase.b)
        if result != testCase.result {
            t.Errorf("Adding %d and %d doesn't produce %d, instead it produces \
            %d", testCase.a, testCase.b, testCase.result, result)
        }
    })
}
```

As you can see from the code, you change the iterator variable to something else (`tc`) and make `testCase` a local variable that takes the value of `tc`. When you rerun the test, this should fix the problem:

```
% go test -v -run TestAddWithSubTestAndParallel
=== RUN   TestAddWithSubTestAndParallel
=== RUN   TestAddWithSubTestAndParallel/OneDigit
=== PAUSE TestAddWithSubTestAndParallel/OneDigit
=== RUN   TestAddWithSubTestAndParallel/TwoDigits
=== PAUSE TestAddWithSubTestAndParallel/TwoDigits
=== RUN   TestAddWithSubTestAndParallel/ThreeDigits
=== PAUSE TestAddWithSubTestAndParallel/ThreeDigits
=== CONT  TestAddWithSubTestAndParallel/OneDigit
=== CONT  TestAddWithSubTestAndParallel/ThreeDigits
=== CONT  TestAddWithSubTestAndParallel/OneDigit
    testing_test.go:108: Test case OneDigit with inputs 1 and 2 should produce 3
=== CONT  TestAddWithSubTestAndParallel/ThreeDigits
    testing_test.go:108: Test case ThreeDigits with inputs 100 and -1 should
        produce 99
=== CONT  TestAddWithSubTestAndParallel/TwoDigits
    testing_test.go:108: Test case TwoDigits with inputs 12 and 30 should produce
        42
--- PASS: TestAddWithSubTestAndParallel (0.00s)
    --- PASS: TestAddWithSubTestAndParallel/TwoDigits (0.50s)
    --- PASS: TestAddWithSubTestAndParallel/OneDigit (0.50s)
    --- PASS: TestAddWithSubTestAndParallel/ThreeDigits (0.50s)
PASS
ok      github.com/sausheong/gocookbook/ch18_testing    0.699s
```

18.6 Generating Random Test Inputs for Tests

Problem

You want to generate random test data for running your test functions.

Solution

Use fuzzing, which is an automated testing technique to generate random test data for your test functions.

Discussion

Fuzzing, or fuzz testing, is an automated testing technique that generates random, unexpected data for your program in order to detect bugs. Fuzzing has been around for quite a while; the first paper on fuzzing was published in 1990. Go has had fuzzing libraries for a while as well, but in Go 1.18, fuzzing was added as a feature. The feature was added as part of the `go test` tool as well as the standard library.

You can use fuzzing to test the max heap implementation you first saw in Recipe 14.5. For simplicity's sake, in the following code snippet some of the functions have been removed, keeping only those that are relevant here:

```
type Heap struct {
        elements []int
}

func (h *Heap) Push(ele int) {
        h.elements = append(h.elements, ele)
        i := len(h.elements) - 1
        for ; h.elements[i] > h.elements[parent(i)]; i = parent(i) {
                h.swap(i, parent(i))
        }
}

func (h *Heap) Pop() (ele int) {
        ele = h.elements[0]
        h.elements[0] = h.elements[len(h.elements)-1]
        h.elements = h.elements[:len(h.elements)-1]
        h.rearrange(0)
        return
}

func (h *Heap) rearrange(i int) {
        largest := i
        left, right, size := leftChild(i), rightChild(i), len(h.elements)
        if left < size && h.elements[left] > h.elements[largest] {
                largest = left
        }
        if right < size && h.elements[right] > h.elements[largest] {
                largest = right
        }
        if largest != i {
                h.swap(i, largest)
                h.rearrange(largest)
        }
}
```

Fuzzing is useful because it automates input data into your test functions such that it tests unexpected cases. If you were to test the max heap implementation discussed earlier, this is a typical test function you might write, which will test the `Push` and `Pop` functions:

```
func TestHeap(t *testing.T) {
        var h *Heap = &Heap{}
        h.elements = []int{452, 23, 6515, 55, 313, 6}
        h.Build()

        testCases := []int{51, 634, 9, 8941, 354}
        for _, tc := range testCases {
                h.Push(tc)
                // make a copy of the elements in the slice and sort it in
                // descending order
                elements := make([]int, len(h.elements))
                copy(elements, h.elements)
```

```
                sort.Slice(elements, func(i, j int) bool {
                        return elements[i] > elements[j]
                })
                // pop the heap and check if the top of heap is the largest
                // element
                popped := h.Pop()
                if elements[0] != popped {
                        t.Errorf("Top of heap %d is not the one popped %d\n heap\
                                is %v",
                                elements[0], popped, elements)
                }
        }
}
```

First, create a max heap and prepopulate the heap with data. Next, use a set of test cases (which are just a bunch of integers), and push them into the heap. You want to pop the heap, which will give you the largest integer in the heap.

To check if this is the case, take the slice of elements that is the data for the heap and sort it in descending order. The first element of the slice is the largest integer and should be the same as the integer you get from popping the heap.

When you run the test function with these test cases, everything works fine:

```
% go test -run=TestHeap -v
=== RUN   TestHeap
--- PASS: TestHeap (0.00s)
PASS
ok      github.com/sausheong/gocookbook/ch18_testing    0.229s
```

As you can see, you test only with this input data into the heap. This is where fuzzing comes in. In Go 1.18, fuzzing was introduced in the `go test` toolset. Fuzz tests are added as fuzz functions in the same *_test.go* files you use for the test functions.

Each fuzz function must start with `Fuzz`, similar to how test functions start with `Test`; and each takes only one parameter, which is a pointer to `testing.F`.

There are two parts to creating a fuzz function:

- Seeding the input to the fuzz function using the `f.Add` function.
- Running the fuzz test itself by calling the `f.Fuzz` function and passing it a *fuzz target*, which is a function that has a pointer to the `testing.T` parameter, as well as a set of *fuzzing arguments*.

Take a look at how you can convert your test function to a fuzz function:

```
func FuzzHeap(f *testing.F) {
        var h *Heap = &Heap{}
        h.elements = []int{452, 23, 6515, 55, 313, 6}
        h.Build()
```

```
        testCases := []int{51, 634, 9, 8941, 354}
        for _, tc := range testCases {
                f.Add(tc)
        }

        f.Fuzz(func(t *testing.T, i int) {
                h.Push(i)
                // make a copy of the elements in the slice and sort it in
                // descending order
                elements := make([]int, len(h.elements))
                copy(elements, h.elements)
                sort.Slice(elements, func(i, j int) bool {
                        return elements[i] > elements[j]
                })
                // pop the heap and check if the top of heap is the largest
                // element
                popped := h.Pop()
                if elements[0] != popped {
                        t.Errorf("Top of heap %d is not the one popped %d\n heap\
                                is %v", elements[0], popped, elements)
                }
        })
}
```

You create a function named `FuzzHeap` that accepts a pointer to `testing.F`. In this function, you start off by setting up the max heap as before. Then you take the test cases and add them to the *seed corpus*, the collection of seed input for the fuzz tests, using `f.Add`.

The fuzz target has a pointer `testing.T` as well as a single integer. The fuzzing arguments must be the same and also in the same sequence as the parameters you pass into `f.Add` as you register the inputs into the seed corpus. In your fuzz function, you pass a single integer into the `f.Add` function, so you will have only a single integer as the fuzzing argument.

The fuzz target body is the same as the earlier test function, and you're done! Run it!

To run a fuzz function, you need to use the `-fuzz` flag, passing it a part of the function name (or simply a period to indicate everything). You can also pass in a `-fuzztime` parameter to indicate how long you want to run the fuzz function, because fuzz functions will run forever if they can't find any bugs!

```
% go test -v -fuzz=Heap -fuzztime=30s
=== RUN   TestHeap
--- PASS: TestHeap (0.00s)
=== FUZZ  FuzzHeap
fuzz: elapsed: 0s, gathering baseline coverage: 0/1484 completed
fuzz: elapsed: 0s, gathering baseline coverage: 1484/1484 completed, now fuzzing
with 10 workers
fuzz: elapsed: 3s, execs: 692916 (230887/sec), new interesting: 1 (total: 1485)
fuzz: elapsed: 6s, execs: 1343416 (216901/sec), new interesting: 1 (total: 1485)
```

```
fuzz: elapsed: 9s, execs: 2078265 (244901/sec), new interesting: 1 (total: 1485)
fuzz: elapsed: 12s, execs: 2827429 (249737/sec), new interesting: 1 (total: 1485)
fuzz: elapsed: 15s, execs: 3527717 (233462/sec), new interesting: 1 (total: 1485)
fuzz: elapsed: 18s, execs: 4256457 (242874/sec), new interesting: 1 (total: 1485)
fuzz: elapsed: 21s, execs: 5014656 (252735/sec), new interesting: 1 (total: 1485)
fuzz: elapsed: 24s, execs: 5757659 (247697/sec), new interesting: 1 (total: 1485)
fuzz: elapsed: 27s, execs: 6447953 (230105/sec), new interesting: 1 (total: 1485)
fuzz: elapsed: 30s, execs: 7175096 (242388/sec), new interesting: 1 (total: 1485)
fuzz: elapsed: 30s, execs: 7175096 (0/sec), new interesting: 1 (total: 1485)
--- PASS: FuzzHeap (30.30s)
PASS
ok      github.com/sausheong/gocookbook/ch18_testing    30.935s
```

The first line indicates that the baseline coverage is gathered by executing the test with the seed corpus and the generated corpus before fuzzing begins. If the test doesn't work in the first place, there's no point doing fuzzing.

The number of workers indicates how many fuzz targets are run in parallel. You can actually specify this using the `-parallel` flag, but if you leave it empty, it will use `GOMAXPROCS`, which by default is the number of cores available.

In the following lines, `elapsed` shows how long the fuzzing has been running, `execs` shows the total number of inputs that have been run against the fuzz target, while `new interesting` shows how many inputs have expanded the code coverage beyond existing corpora, with the size of the entire corpus.

The fuzz function itself can be run as a normal test function with the seed corpus. If you run it with `go test` as you would any test function, you should get these results:

```
% go test -run=FuzzHeap -v
=== RUN   FuzzHeap
=== RUN   FuzzHeap/seed#0
=== RUN   FuzzHeap/seed#1
=== RUN   FuzzHeap/seed#2
=== RUN   FuzzHeap/seed#3
=== RUN   FuzzHeap/seed#4
--- PASS: FuzzHeap (0.00s)
    --- PASS: FuzzHeap/seed#0 (0.00s)
    --- PASS: FuzzHeap/seed#1 (0.00s)
    --- PASS: FuzzHeap/seed#2 (0.00s)
    --- PASS: FuzzHeap/seed#3 (0.00s)
    --- PASS: FuzzHeap/seed#4 (0.00s)
PASS
ok      github.com/sausheong/gocookbook/ch18_testing    0.246s
```

As you can see, you have five runs of the fuzz target against the five seed inputs in the seed corpus, and all of them pass.

This is all good, but it doesn't really show how fuzzing helps make the software more robust. A simple example can show this. Change your `rearrange` function a

bit. Instead of comparing `h.elements[left]` you compare `h.elements[left-1]`. It's a small change that can result in an error, and it can easily go undetected:

```
func (h *Heap) rearrange(i int) {
    ...
        if left < size && h.elements[left-1] > h.elements[largest] {
                largest = left
        }
    ...
}
```

To prove this, run it against your `TestHeap` test function. You should see that it runs perfectly well and the test case passes. Now run it against the `FuzzHeap` fuzz function:

```
% go test -v -fuzz=Heap -fuzztime=30s
=== RUN   TestHeap
--- PASS: TestHeap (0.00s)
=== FUZZ  FuzzHeap
fuzz: elapsed: 0s, gathering baseline coverage: 0/1484 completed
fuzz: elapsed: 0s, gathering baseline coverage: 19/1484 completed
--- FAIL: FuzzHeap (0.28s)
    --- FAIL: FuzzHeap (0.00s)
        testing_test.go:260: Top of heap 313 is not the one popped 158
             heap is [313 158 55 23 6 -327 -349]

    Failing input written to testdata/fuzz/FuzzHeap/03b1c861389a9c041082690dc8b
        25528f6ff6debab2a7fc99524a738895bea1f
    To re-run:
    go test -run=FuzzHeap/03b1c861389a9c041082690dc8b25528f6ff6debab2a7fc99524a
        738895bea1f
FAIL
exit status 1
FAIL    github.com/sausheong/gocookbook/ch18_testing    0.540s
```

As you can see, it fails at the baseline coverage, and the element that was popped from the heap wasn't the maximum. You can also see that the input to the failed test case is written to a test data file. If you open it, you should see something like this:

```
go test fuzz v1
int(-349)
```

And if you run the `FuzzHeap` function as a normal test function, you will immediately see that the other test cases pass with the other input, but with `-349` the max heap doesn't work any more:

```
% go test -run=FuzzHeap -v
=== RUN   FuzzHeap
=== RUN   FuzzHeap/seed#0
=== RUN   FuzzHeap/seed#1
=== RUN   FuzzHeap/seed#2
=== RUN   FuzzHeap/seed#3
=== RUN   FuzzHeap/seed#4
=== RUN   FuzzHeap/03363930589906b56680eea723dd29e2744bd87e28b0995dd65209094
```

```
            ef3080d
        testing_test.go:260: Top of heap 313 is not the one popped 51
             heap is [313 55 51 48 23 9 6]
    --- FAIL: FuzzHeap (0.00s)
        --- PASS: FuzzHeap/seed#0 (0.00s)
        --- PASS: FuzzHeap/seed#1 (0.00s)
        --- PASS: FuzzHeap/seed#2 (0.00s)
        --- PASS: FuzzHeap/seed#3 (0.00s)
        --- PASS: FuzzHeap/seed#4 (0.00s)
        --- FAIL: FuzzHeap/03363930589906b56680eea723dd29e2744bd87e28b0995dd65209094
            ef3080d (0.00s)
    FAIL
    exit status 1
    FAIL    github.com/sausheong/gocookbook/ch18_testing    0.475s
```

You can imagine this can be pretty hard to detect! If you fix the code, you can run the same `FuzzHeap` test again and see that it has passed all the tests, including a regression one that was automatically generated from a failed fuzz test:

```
% go test -run=FuzzHeap -v
=== RUN   FuzzHeap
=== RUN   FuzzHeap/seed#0
=== RUN   FuzzHeap/seed#1
=== RUN   FuzzHeap/seed#2
=== RUN   FuzzHeap/seed#3
=== RUN   FuzzHeap/seed#4
=== RUN   FuzzHeap/03b1c861389a9c041082690dc8b25528f6ff6debab2a7fc99524a
          738895bea1f
--- PASS: FuzzHeap (0.00s)
    --- PASS: FuzzHeap/seed#0 (0.00s)
    --- PASS: FuzzHeap/seed#1 (0.00s)
    --- PASS: FuzzHeap/seed#2 (0.00s)
    --- PASS: FuzzHeap/seed#3 (0.00s)
    --- PASS: FuzzHeap/seed#4 (0.00s)
    --- PASS: FuzzHeap/03b1c861389a9c041082690dc8b25528f6ff6debab2a7fc99524a
        738895bea1f (0.00s)
PASS
ok      github.com/sausheong/gocookbook/ch18_testing    0.283s
```

Fuzzing is a powerful tool. However, it can be pretty expensive to run, especially in an automated continuous integration pipeline, since it can be CPU intensive.

18.7 Measuring Test Coverage

Problem

You want to know how much of the program code has been covered by tests.

Solution

Use the test coverage feature that is built into the `go test` tool.

Discussion

Test coverage is a metric that measures the amount of testing done on the codebase. It's immediately obvious that this is an important metric as it gives you a level of confidence in your code. In fact, test coverage was one of the first metrics for systematic software testing and was first mentioned in *Communications of the ACM* in 1963.

Go provides a test coverage feature in the Go 1.2 release, which like all other test features is integrated within the `go test` tool.

In this recipe, you'll use the code for all the other recipes in this chapter. We have written test functions for them, so it's time to check how well you have covered them.

The simplest way to use test coverage is simply to use the `-cover` flag:

```
% go test -cover
PASS
coverage: 94.6% of statements
ok      github.com/sausheong/gocookbook/ch18_testing    5.428s
```

That's pretty simple! You can see that your coverage is pretty good. However, this doesn't tell you more information as to what you missed. To do that, you need a coverage profile. You can do that by using the `-coverprofile` flag and specifying a file for the output:

```
% go test -coverprofile=coverage.out
PASS
coverage: 94.6% of statements
ok      github.com/sausheong/gocookbook/ch18_testing    5.899s
```

As you can see, the output on the command line is still the same, but now a *coverage.out* file is produced, which contains something like this (truncated, as the file can be long):

```
mode: set
github.com/sausheong/gocookbook/ch18_testing/testing.go:12.24,15.2 2 1
github.com/sausheong/gocookbook/ch18_testing/testing.go:18.33,19.33 1 1
github.com/sausheong/gocookbook/ch18_testing/testing.go:19.33,21.35 2 1
github.com/sausheong/gocookbook/ch18_testing/testing.go:21.35,24.4 2 1
github.com/sausheong/gocookbook/ch18_testing/testing.go:29.50,33.28 4 1
github.com/sausheong/gocookbook/ch18_testing/testing.go:38.2,39.16 2 1
...
```

With this coverage profile, next you can use the Go `cover` tool to find out more information. As a start, you can use the `-func` flag to find out how well each function is covered:

```
% go tool cover -func=coverage.out
github.com/sausheong/gocookbook/ch18_testing/testing.go:12:  Add        100.0%
github.com/sausheong/gocookbook/ch18_testing/testing.go:18:  flip       100.0%
github.com/sausheong/gocookbook/ch18_testing/testing.go:29:  save       90.9%
github.com/sausheong/gocookbook/ch18_testing/testing.go:47:  load       85.7%
github.com/sausheong/gocookbook/ch18_testing/testing.go:72:  Push       100.0%
github.com/sausheong/gocookbook/ch18_testing/testing.go:80:  Pop        100.0%
github.com/sausheong/gocookbook/ch18_testing/testing.go:88:  rearrange  100.0%
github.com/sausheong/gocookbook/ch18_testing/testing.go:103: Build      100.0%
github.com/sausheong/gocookbook/ch18_testing/testing.go:109: swap       100.0%
github.com/sausheong/gocookbook/ch18_testing/testing.go:113: parent     100.0%
github.com/sausheong/gocookbook/ch18_testing/testing.go:117: leftChild  100.0%
github.com/sausheong/gocookbook/ch18_testing/testing.go:121: rightChild 100.0%
total:  (statements)    94.6%
```

This is much better, as you can now see the coverage per function. Another way to represent the coverage is an HTML coverage report, which provides source code–level details on the coverage. To do so, run this on the command line:

```
% go tool cover -html=coverage.out
```

This should pop up an HTML page that looks something like Figure 18-1, which shows the coverage report generated by the Go code coverage tool.

The covered code is in green, the uncovered code is in red, and the uninstrumented code is in gray. This gives us a very clear indication of which part of our code is covered. (Note that the colors will not show up in the print edition of this book.)

For even more fine-grained information about the coverage, you can use the `-covermode` flag and set the mode to the `count` setting:

```
% go test -covermode=count -coverprofile=count.out
PASS
coverage: 94.6% of statements
ok      github.com/sausheong/gocookbook/ch18_testing    5.980s
```

This creates another file, *count.out*, that looks something like this:

```
mode: count
github.com/sausheong/gocookbook/ch18_testing/testing.go:12.24,15.2 2 16
github.com/sausheong/gocookbook/ch18_testing/testing.go:18.33,19.33 1 4
github.com/sausheong/gocookbook/ch18_testing/testing.go:19.33,21.35 2 1284
github.com/sausheong/gocookbook/ch18_testing/testing.go:21.35,24.4 2 308160
github.com/sausheong/gocookbook/ch18_testing/testing.go:29.50,33.28 4 2
github.com/sausheong/gocookbook/ch18_testing/testing.go:38.2,39.16 2 2
...
```

To view the HTML coverage report in count mode, you can do this:

```
% go tool cover -html=count.out
```

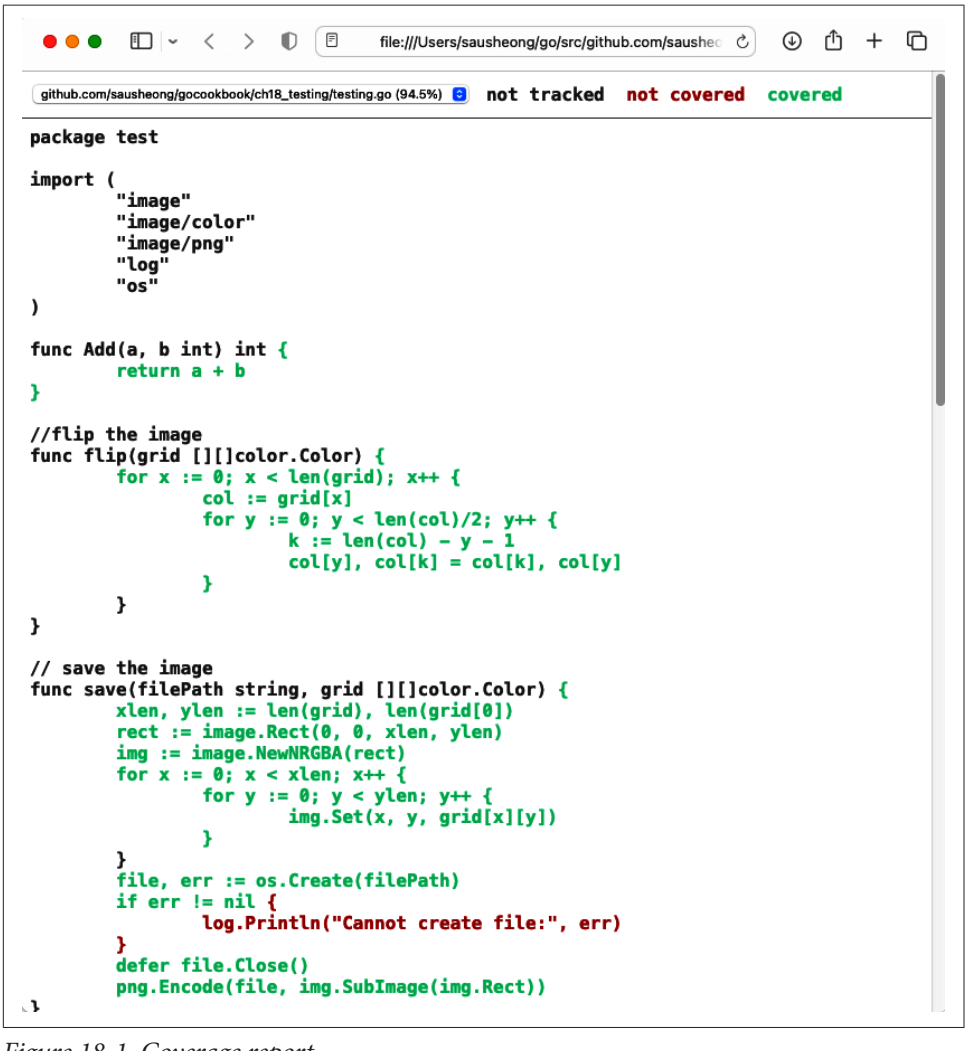

Figure 18-1. Coverage report

You should see an HTML page that looks like Figure 18-2, which shows the coverage report in count mode.

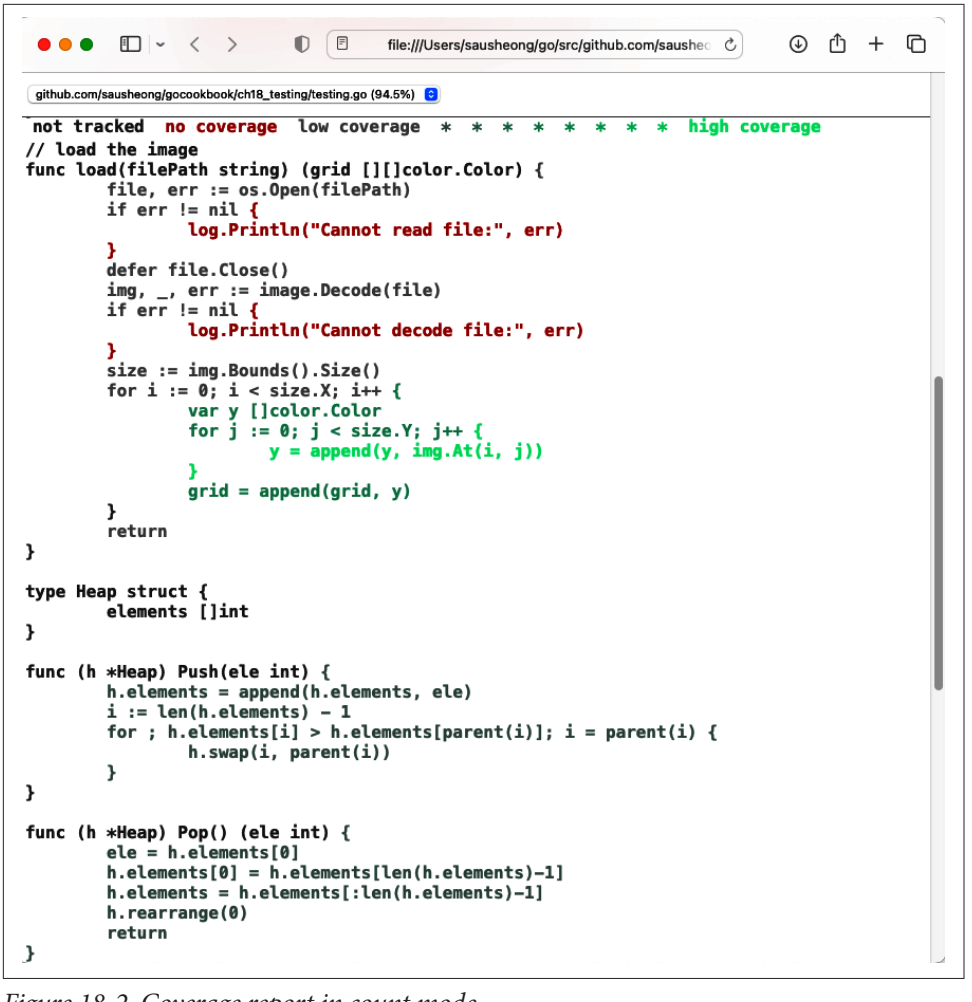

Figure 18-2. Coverage report in count mode

What's interesting in this report is that you can see how well-covered the code is, shown in the different shades of green. Note that this also is not going to show in the print edition, so I encourage you to try this yourself to see what it looks like.

18.8 Testing a Web Application or a Web Service

Problem

You want to do unit testing on a web application or a web service.

Solution

Use the `httptest.NewRecorder` function to create an `httptest.ResponseRecorder` that can be used to record what's been written to the `http.ResponseWriter`. This can then be used to test the response.

Discussion

Web applications and web services are the most popular type of programs written in Go, so obviously you need to be able to test them. The `httptest` package provides a way to do this.

The handler function is the function that is called when a request is received. The handler function writes the response to the `http.ResponseWriter`, which is then passed back to the client. This approach focuses on testing the handler function that is used to handle the request, which covers most of what you need. Specific functions can be tested using the normal approach, but testing the handler is a bit tricky because you need to create an HTTP server to test it.

The `httptest.NewRecorder` function creates an `httptest.ResponseRecorder` that implements the `http.ResponseWriter` interface. This can be used to record what's been written to the `http.ResponseWriter` so that you can test the response:

```
Let's say we have this handler function we want to test.

func hello(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintf(w, "Hello World!")
}
```

You can test this using the `httptest` package like this:

```
func TestHttpHello(t *testing.T) {
        http.HandleFunc("/hello", hello)
        writer := httptest.NewRecorder()
        request, _ := http.NewRequest("GET", "/hello", nil)
        http.DefaultServeMux.ServeHTTP(writer, request)

        if writer.Code != http.StatusOK {
                t.Errorf("Response code is %v", writer.Code)
        }

        if expected, actual := "Hello World!", writer.Body.String();
                expected != actual {
                t.Errorf("Response body is %v", actual)
        }
}
```

First, you need to register the handler function with the `http.DefaultServeMux` using the `http.HandleFunc` function. This is the same way you would register the handler function normally. Then you create an `httptest.ResponseRecorder` using

the `httptest.NewRecorder` function. This implements the `http.ResponseWriter` interface.

Next, you need to create the client to emulate someone sending an HTTP request to the server. Use the `http.NewRequest` function to create a `GET` request to the `/hello` URL. With this, you have both the `http.ResponseWriter` and `http.Request` that you can pass to `ServeHTTP` to dispatch the request to the handler function that matches the URL. Remember, when you registered the handler function using `http.HandleFunc`, you actually registered the function to `http.DefaultServeMux`. This is the same multiplexer you are using to dispatch the request.

Once the request is dispatched, the response is written to the `httptest.ResponseRecorder` that you created. You can then test the response code and the response body to make sure that the response is what you expect.

The `httptest.ResponseRecorder` has a `Code` field that contains the HTTP status code that was written to the `http.ResponseWriter`. You can test this to make sure it's the expected value. The `httptest.ResponseRecorder` also has a `Body` field that contains the response body that was written to the `http.ResponseWriter`. You can test this as well to make sure it's the expected value. You can also test for other parts of the response, such as the headers, cookies, etc.

CHAPTER 19

Benchmarking Recipes

19.0 Introduction

Performance testing is an important part of software testing. It's nonfunctional, meaning it doesn't test whether the software does what it's supposed to do. Instead, it tests how well the software performs in terms of stability, speed, and scalability under a given workload.

In Go, performance testing is done with the same testing tools as functional testing, using the `go test` tool and the `testing` standard library.

19.1 Automating Performance Tests

Problem

You want to automate the performance testing of a function.

Solution

Create a benchmark function and use the `go test` tool to run it.

Discussion

Go provides a minimalist set of built-in tools for testing with the `go test` command-line tool and the `testing` package. These tools are used for both functional and performance testing. In this recipe, we'll focus on performance testing.

As in Chapter 18, you'll be using a simple `Add` function and try to do performance testing on it:

```
package test

func Add(a, b int) int {
        return a + b
}
```

The test flow for performance testing is the same as functional testing. First you define a set of benchmark functions in a file that has a general naming convention of *_test.go*. In this case, put your benchmark functions in a *benchmark_test.go* file. As in functional tests, the name of the package the file is in must be the same as the one with the function you want to do performance testing on:

```
func BenchmarkAdd(b *testing.B) {
        for i := 0; i < b.N; i++ {
                Add(1, 2)
        }
}
```

Each benchmark function starts with "Benchmark," followed by "<*Xxx*>," and you use camel case to describe what you're testing. There must be only a single input parameter to the benchmark function, which is a pointer to `testing.B`. `B` is a struct that is passed to benchmark functions to manage benchmark timing and specify how many iterations to run. It is very similar to the `T` you saw in test functions used in functional testing.

The main part of the body of any benchmark function is the loop, which normally repeats `b.N` times, each time executing the statements you want to run performance tests on. You need to write this loop. You can see how this works by running the performance test:

```
% go test -bench=.
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch19_benchmarking
BenchmarkAdd-10         1000000000               0.4093 ns/op
PASS
ok      github.com/sausheong/gocookbook/ch19_benchmarking       0.907s
```

You run the performance test using the `go test` tool and the `-bench` flag and passing it the argument `.`. This will tell the tool to run all benchmark functions. If you want to run only certain benchmark functions, you need to use a regular expression to match the benchmarks you want to run.

The benchmark results tell you several things, and up to the `cpu` line only tells you the environment you are running the tests with. The line that starts with `BenchmarkAdd-10` tells you the name of the benchmark function and the number

of cores that the benchmark was run with. The next value is the number of iterations (in this case, this was run 1 billion times!), and the last is the time it took (on average) to run the benchmark. In this example, it took, on average, 0.4093 nanoseconds to run `Add`.

If you run the performance test again, you might discover that the timing might be different. To satisfy yourself that the performance is not a one-off, you can do that a few times. Or you can use the `-count` flag to specify the number of times to run the same performance test:

```
% go test -bench=. -count=5
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch19_benchmarking
BenchmarkAdd-10         1000000000               0.4106 ns/op
BenchmarkAdd-10         1000000000               0.4100 ns/op
BenchmarkAdd-10         1000000000               0.4099 ns/op
BenchmarkAdd-10         1000000000               0.4108 ns/op
BenchmarkAdd-10         1000000000               0.4096 ns/op
PASS
ok      github.com/sausheong/gocookbook/ch19_benchmarking       2.451s
```

19.2 Running Only Performance Tests

Problem

You want to run only performance tests without running the functional tests.

Solution

When you have functional tests along with performance tests in the same file, both kinds of tests are going to be run. To just run performance tests, use the `-run` flag to filter out functional tests.

Discussion

Benchmark functions and test functions are in the same file, and when you run performance tests, all the functions are run, including test functions. Conversely, this is not true; if you run functional tests, performance tests are not run unless you add the `-bench` flag. If you only want to run performance tests, this can be quite irritating.

Let's say you have this test function in the same file:

```
func TestAdd(t *testing.T) {
        result := Add(1, 2)
        if result != 3 {
                t.Error("Adding 1 and 2 doesn't produce 3")
        } else {
```

```
                t.Log("Adding 1 and 2 results in", result)
        }
}
```

When you try to run the performance test, you will get this:

```
% go test -v -bench=.
=== RUN   TestAdd
    benchmark_test.go:16: Adding 1 and 2 results in 3
--- PASS: TestAdd (0.00s)
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch19_benchmarking
BenchmarkAdd
BenchmarkAdd-10         1000000000               0.4099 ns/op
PASS
ok      github.com/sausheong/gocookbook/ch19_benchmarking       0.901s
```

This looks harmless if there is just one test, but if there are a number of them and they take some time to run, it can be quite painful. You can use the `-run` flag to filter out functional tests that match the filter. If you put a filter that doesn't match anything, no functional tests will be run:

```
% go test -v -bench=. -run=XXX
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch19_benchmarking
BenchmarkAdd
BenchmarkAdd-10         1000000000               0.4109 ns/op
PASS
ok      github.com/sausheong/gocookbook/ch19_benchmarking       0.634s
```

19.3 Avoiding Test Fixtures in Performance Tests

Problem

You want to customize the performance tests to avoid benchmarking test fixtures.

Solution

You can start, stop, and reset the benchmark timers using the `StartTimer`, `StopTimer`, and `ResetTimer`, respectively. This will allow you the flexibility to avoid test fixtures.

Discussion

As with functional tests, you will sometimes need to set up test fixtures before running the performance test. Here is an example where you want to take an image file and flip it. In this example, you want to flip a PNG-format image of the Mona Lisa.

The algorithm is easy, and you first load the image file into a 2D grid of pixels (where a pixel is represented by a `color.Color` struct). (For more information about the algorithm, see Recipe 15.3.)

The specific code you want to test is the `flip` function that takes the grid and flips the pixels in them:

```
//flip the image
func flip(grid [][]color.Color) {
        for x := 0; x < len(grid); x++ {
                col := grid[x]
                for y := 0; y < len(col)/2; y++ {
                        k := len(col) - y - 1
                        col[y], col[k] = col[k], col[y]
                }
        }
}
```

The test fixture you need to set up is to take a PNG file and load it into a grid to get it ready for testing:

```
func BenchmarkFlip(b *testing.B) {
        grid := load("monalisa.png")
        for i := 0; i < b.N; i++ {
                flip(grid)
        }
}
```

Let's run this benchmark function:

```
% go test -v -bench=Flip -run=XXX
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch19_benchmarking
BenchmarkFlip
BenchmarkFlip-10            6492            184067 ns/op
PASS
ok      github.com/sausheong/gocookbook/ch19_benchmarking       1.538s
```

You can see that it takes around 184,067 nanoseconds (or around 184 microseconds) on average to do this. But wait, there is an issue here because the benchmark timer starts at the beginning of the benchmark function, which means this timing includes the setup activity of loading up the file. To overcome this, you need to reset the timer after doing the setup activities by calling `b.ResetTimer()` after loading the PNG file:

```
func BenchmarkFlip(b *testing.B) {
        grid := load("monalisa.png")
        b.ResetTimer()
        for i := 0; i < b.N; i++ {
                flip(grid)
        }
}
```

Run it again and see what happens:

```
% go test -v -bench=Flip -run=XXX
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch19_benchmarking
BenchmarkFlip
BenchmarkFlip-10            6618            181478 ns/op
PASS
ok      github.com/sausheong/gocookbook/ch19_benchmarking       2.338s
```

The timing is now about 181 microseconds instead, about 3 microseconds difference.

This is great for setting up before you get into the benchmarking loop. For example, you need to actually do something every iteration and not just once before the start of the loop:

```
func BenchmarkLoadAndFlip(b *testing.B) {
        for i := 0; i < b.N; i++ {
                grid := load("monalisa.png")
                flip(grid)
        }
}
```

Take a look at the timing:

```
% go test -v -bench=LoadAndFlip -run=XXX
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch19_benchmarking
BenchmarkLoadAndFlip
BenchmarkLoadAndFlip-10               69          14613379 ns/op
PASS
ok      github.com/sausheong/gocookbook/ch19_benchmarking       1.232s
```

This took about 14.6 milliseconds. To ignore the timing for loading the image file, you need to stop the timer before calling `load` and start the timer afterward. For this, you can use the `StopTimer` and `StartTimer` to control which parts of the iteration you don't want to benchmark:

```
func BenchmarkLoadAndFlip(b *testing.B) {
        for i := 0; i < b.N; i++ {
                b.StopTimer()
                grid := load("monalisa.png")
                b.StartTimer()
                flip(grid)
        }
}
```

Run it again and see the results:

```
% go test -v -bench=LoadAndFlip -run=XXX
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch19_benchmarking
BenchmarkLoadAndFlip
BenchmarkLoadAndFlip-10             1540            672674 ns/op
PASS
ok      github.com/sausheong/gocookbook/ch19_benchmarking       23.672s
```

The timing for the `flip` function is about 0.67 milliseconds. However, the entire performance test took a lot longer, almost 24 seconds! This is because the `flip` function is much faster than the `load` function, and therefore you could do many more iterations (1,540 compared with 69 earlier). However, even though you are not considering the timing for the `load` function in the performance test, it still executes and takes much longer.

19.4 Changing the Timing for Running Performance Tests

Problem

You want to run performance tests for a specific duration or a specific number of iterations.

Solution

You can increase the minimum duration the benchmarks should run or increase the number of iterations using the `-benchtime` flag.

Discussion

By default, Go iterates several times in the benchmark function such that it takes roughly 1 second.

For example, for this benchmark function:

```
func BenchmarkLoadAndFlip(b *testing.B) {
        for i := 0; i < b.N; i++ {
                b.StopTimer()
                grid := load("monalisa.png")
                b.StartTimer()
                flip(grid)
        }
}
```

when you run the performance test, you get this:

```
% go test -v -bench=LoadAndFlip -run=XXX
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch19_benchmarking
BenchmarkLoadAndFlip
BenchmarkLoadAndFlip-10             1540            672674 ns/op
PASS
ok      github.com/sausheong/gocookbook/ch19_benchmarking       23.672s
```

The results tell you that the benchmark function ran 1,540 iterations, and the timing for each loop is 672,674 nanoseconds or about 0.67 milliseconds. If you multiply 1,540 by 672,674, you get 1,035,917,960 nanoseconds or about 1 second.

However, you can see that it runs for a long time because the `load` function, while it's not included in the timing calculations, takes quite some time. This is because although the timing is about 1 second, the real running time for the performance test is about 24 seconds!

To run this faster, you can specify the timing to run using the `-benchtime` flag:

```
% go test -v -bench=LoadAndFlip -run=XXX -benchtime=100ms
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch19_benchmarking
BenchmarkLoadAndFlip
BenchmarkLoadAndFlip-10              226            687917 ns/op
PASS
ok      github.com/sausheong/gocookbook/ch19_benchmarking       4.789s
```

In this case, you are saying you want to run the benchmark in 100 milliseconds instead of 1 second (which is about 10 times less). Correspondingly the iterations reduced to 226, and the overall timing dropped to 4.8 seconds. The timing isn't the most accurate, though. As you can see, if you multiply 226 by 687,917, you get around 155.5 milliseconds and not 100 milliseconds.

You can also use the `-benchtime` flag to specify the number of times the iterator should run, using the argument `Nx`, where `N` is the number of times to run:

```
% go test -v -bench=LoadAndFlip -run=XXX -benchtime=100x
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch19_benchmarking
BenchmarkLoadAndFlip
BenchmarkLoadAndFlip-10              100            712695 ns/op
PASS
ok      github.com/sausheong/gocookbook/ch19_benchmarking       1.932s
```

You can see here that you're running 100 iterations in the loop.

19.5 Running Multiple Performance Test Cases

Problem

You want to run multiple benchmark test cases with table-driven tests like in functional testing.

Solution

Create sub-benchmarks and run each test case with a sub-benchmark.

Discussion

You have seen table-driven tests in Recipe 18.2. They provide a means to run different test cases for a test function and a better way to organize tests together. In the same way, you also want to do table-driven performance tests.

Say you have this recursive Fibonacci series function that you want to test its performance on:

```
func fibonacci(n int) int {
        if n <= 1 {
                return n
        }
        return fibonacci(n-1) + fibonacci(n-2)
}
```

The larger the number you pass into the function, the more times it will recurse and the longer it will take. This means the performance of the function depends on the number you pass into the function.

Take a look at running a performance test on `fibonacci` with the parameter 5:

```
func BenchmarkFibonacci5(b *testing.B) {
        for i := 0; i < b.N; i++ {
                fibonacci(5)
        }
}
```

If you run it, you will see something like this:

```
% go test -run=XXX -bench=Fibonacci5
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch19_benchmarking
BenchmarkFibonacci5-10          43522675                27.53 ns/op
PASS
ok      github.com/sausheong/gocookbook/ch19_benchmarking       1.836s
```

However, table-driven performance testing the same way you did functional testing is not possible prior to Go 1.7. This is because a benchmark function provides only a single set of performance results, whatever you do. You can create multiple benchmark functions, but that's not table-driven testing anymore.

Subtests were introduced in Go 1.7, and this included the feature to run sub-benchmarks as well. This provided the capability to finally do table-driven performance testing:

```
func BenchmarkFibonacciWithSubBenchmark(b *testing.B) {
        testCases := []struct {
                name string
                n    int
        }{
                {"Fibonacci-1", 1},
                {"Fibonacci-5", 5},
                {"Fibonacci-10", 10},
                {"Fibonacci-20", 20},
                {"Fibonacci-30", 30},
        }
        for _, testCase := range testCases {
                testCase := testCase
                b.Run(testCase.name, func(b *testing.B) {
                        for i := 0; i < b.N; i++ {
                                fibonacci(testCase.n)
                        }
                })
        }
}
```

If you now run this test case you should see this:

```
% go test -run=XXX -bench=SubBenchmark
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch19_benchmarking
BenchmarkFibonacciWithSubBenchmark/Fibonacci-1-10       440615576        2.735 ns/op
BenchmarkFibonacciWithSubBenchmark/Fibonacci-5-10        42677919        27.86 ns/op
BenchmarkFibonacciWithSubBenchmark/Fibonacci-10-10        3598915        332.4 ns/op
BenchmarkFibonacciWithSubBenchmark/Fibonacci-20-10          29084        41173 ns/op
BenchmarkFibonacciWithSubBenchmark/Fibonacci-30-10            236      5069878 ns/op
PASS
ok      github.com/sausheong/gocookbook/ch19_benchmarking       8.161s
```

19.6 Comparing Performance Test Results

Problem

You want to compare performance test results to see if your code changes have made improvements.

Solution

Use benchstat to compare different results from the performance tests.

Discussion

Running performance tests requires a certain consistency in the environment while the tests are being run. While it seems like a reasonably simple requirement (just don't do anything else while you're running performance tests), it's often less straightforward than you might think. For example, if the test is running a while, you might switch to browsing the web or reading your emails, and that will take up networking, memory, and CPU time. Even if you don't do anything, thermal scaling or the garbage collector or system updates, or some other background processes might suddenly kick in.

As a result, the performance test numbers always have a margin of error. Obviously, the lower the margin of error, the more reliable the numbers would be. This is why you want to run the same test multiple times using the `-count` flag.

Take a look at running the benchmark on the `flip` method from Recipe 19.3 (earlier in this chapter):

```
func BenchmarkFlip(b *testing.B) {
        grid := load("monalisa.png")
        b.ResetTimer()
        for i := 0; i < b.N; i++ {
                flip(grid)
        }
}
```

You want to run it 10 times and analyze the results, so you save the output into a file named *flip.txt*:

```
% go test -bench=BenchmarkFlip -run=XXX -count=10 > flip.txt
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch19_benchmarking
BenchmarkFlip-10            6543            182126 ns/op
BenchmarkFlip-10            6532            182625 ns/op
BenchmarkFlip-10            6614            181799 ns/op
BenchmarkFlip-10            6606            181278 ns/op
BenchmarkFlip-10            6547            182261 ns/op
```

```
BenchmarkFlip-10            6600            181419 ns/op
BenchmarkFlip-10            6607            181435 ns/op
BenchmarkFlip-10            6583            184046 ns/op
BenchmarkFlip-10            6540            184130 ns/op
BenchmarkFlip-10            6562            181718 ns/op
PASS
ok      github.com/sausheong/gocookbook/ch19_benchmarking       14.707s
```

The *benchstat* tool is a simple but useful tool that allows you to analyze the results. You can install it like this:

```
% go install golang.org/x/perf/cmd/benchstat
```

Then, you can run `benchstat` and give it the file with the earlier results:

```
% benchstat flip.txt
name     time/op
Flip-10  182µs ± 1%
```

This tells you that the benchmark function took 182 microseconds, with a variance of 1%. A variance of 1–2% is considered good, and 3–5% is OK, but anything larger than 5% means that not all results are reliable, and you should rerun the performance test with a more stable environment.

It's useful to compare benchmarks to figure out if you have improved your code's performance or if using different methods can improve your code's performance. To show this, you'll use the two ways of encoding JSON from a struct to a JSON string in Go—the `Encode` and the `Marshal` functions.

Let's start with the JSON that you will build your struct from:

```
var jsonString string = `{"name":"Han Solo","height":"180","mass":"80",
"hair_color":"brown","skin_color":"fair","eye_color":"brown","birth_year":
"29BBY","gender":"male","homeworld":"https://swapi.dev/api/planets/22/","films":
["https://swapi.dev/api/films/1/","https://swapi.dev/api/films/2/",
"https://swapi.dev/api/films/3/"],"species":[],"vehicles":[],"starships":
["https://swapi.dev/api/starships/10/","https://swapi.dev/api/starships/22/"],
"created":"2014-12-10T16:49:14.582Z","edited":"2014-12-20T21:17:50.334Z",
"url":"https://swapi.dev/api/people/14/"}`

var jsonBytes []byte = []byte(jsonString)
var person Person
```

You will be creating a `Person` struct as part of the setup by unmarshalling the data into it. Start by creating a benchmark function with JSON marshalling:

```
func BenchmarkWrite(b *testing.B) {
        json.Unmarshal(jsonBytes, &person)
        b.ResetTimer()
        for i := 0; i < b.N; i++ {
                data, _ := json.Marshal(person)
                io.Discard.Write(data)
        }
}
```

Run this benchmark 10 times and save it to a *marshal.txt* file:

```
% go test -bench=Write -run=XXX -count=10 > marshal.txt
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch19_benchmarking
BenchmarkWrite-10         492763              2169 ns/op
BenchmarkWrite-10         550326              2162 ns/op
BenchmarkWrite-10         551032              2154 ns/op
BenchmarkWrite-10         547428              2151 ns/op
BenchmarkWrite-10         546570              2152 ns/op
BenchmarkWrite-10         550374              2155 ns/op
BenchmarkWrite-10         544342              2149 ns/op
BenchmarkWrite-10         550413              2154 ns/op
BenchmarkWrite-10         550292              2155 ns/op
BenchmarkWrite-10         543782              2160 ns/op
PASS
ok      github.com/sausheong/gocookbook/ch19_benchmarking       12.452s
```

Now rewrite your benchmark function using JSON encoding:

```
func BenchmarkWrite(b *testing.B) {
        json.Unmarshal(jsonBytes, &person)
        b.ResetTimer()
        encoder := json.NewEncoder(io.Discard)
        for i := 0; i < b.N; i++ {
                encoder.Encode(person)
        }
}
```

Rerun the performance test, but this time save it to an *encode.txt* file instead:

```
% go test -bench=Write -run=XXX -count=10 > encode.txt
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch19_benchmarking
BenchmarkWrite-10         497305              2112 ns/op
BenchmarkWrite-10         551722              2102 ns/op
BenchmarkWrite-10         557810              2103 ns/op
BenchmarkWrite-10         555228              2102 ns/op
BenchmarkWrite-10         552826              2101 ns/op
BenchmarkWrite-10         558488              2105 ns/op
BenchmarkWrite-10         559686              2098 ns/op
BenchmarkWrite-10         550504              2105 ns/op
BenchmarkWrite-10         554644              2096 ns/op
```

```
BenchmarkWrite-10         560419              2100 ns/op
PASS
ok      github.com/sausheong/gocookbook/ch19_benchmarking       12.270s
```

If you just eyeball both results, you will hardly be able to tell the difference. Maybe JSON encoding edges out JSON marshalling by a bit, but it's hard to tell. Using `benchstat`, however, give you better insights into how much of an improvement.

Run `benchstat` and give both files to it as arguments:

```
% benchstat marshal.txt encode.txt
name      old time/op  new time/op  delta
Write-10  2.16µs ± 1%  2.10µs ± 0%  -2.49%  (p=0.000 n=10+10)
```

With this, you can see that marshalling takes 2.16 microseconds with 1% variance, and encoding takes 2.1 microseconds with negligible variance. The delta of –2.49% tells you the new code (encoding) is 2.49% *faster* than the old code (marshalling).

The `p` here is the p-value, a number used in statistics to tell you how statistically significant it is. A number less than 0.05 is considered statistically significant. The number `n` shows how many of the samples are considered valid. A number `10+10` indicates that 10 samples in the old (marshalling) and 10 samples in the new (encoding) are considered valid. You ran the benchmark function 10 times and have gotten 10 samples, so 100% of the samples are valid. If you get anything less than 90% of samples being valid, you should consider rerunning the performance test.

19.7 Profiling a Program

Problem

You want to find out how your function or program uses CPU time.

Solution

You can profile a program to find out how the program uses CPU time using the `pprof` tool.

Discussion

You use benchmarking to find out the performance of your functions or program so that you can improve it. However, more often than not, you don't even know how your program is using available resources. To know more in-depth and therefore be able to improve, you need to profile your functions or program.

A *profile* is a collection of stack traces that show the sequence of certain events like CPU use, memory allocation, and so on. Just like how Go provides the `go test` tool and the `testing` package for functional and performance testing, Go provides

the `pprof` tool and the `runtime/pprof` package for profiling your functions and programs. It provides a few types of built-in profiles, but we will be discussing the CPU profile in this recipe.

The CPU profile helps you to figure out how much time is spent in processing which parts of your code. When CPU profiling is enabled, the runtime will interrupt itself every 10 milliseconds and record the stack trace. The amount of time the code appears in the profile tells you how much time is spent with that particular line of code.

There are two parts to profiling: creating the profile, which saves it to a file, and then running the `pprof` tool to analyze the profile.

Before you start, look at the example code you'll be profiling on. You'll be profiling the code for resizing an image from Recipe 15.6 using the nearest neighbor interpolation algorithm:

```
func resize(grid [][]color.Color, scale float64) (resized [][]color.Color) {
        xlen, ylen := int(float64(len(grid))*scale), int(float64(len(grid[0]))*
        scale)
        resized = make([][]color.Color, xlen)
        for i := 0; i < len(resized); i++ {
                resized[i] = make([]color.Color, ylen)
        }
        for x := 0; x < xlen; x++ {
                for y := 0; y < ylen; y++ {
                        xp := int(math.Floor(float64(x) / scale))
                        yp := int(math.Floor(float64(y) / scale))
                        resized[x][y] = grid[xp][yp]
                }
        }
        return
}
```

For this to work, you need to load the image from the file, and after resizing, you need to save it back to another file:

```
// load the image from file
func load(filePath string) (grid [][]color.Color) {
        file, err := os.Open(filePath)
        if err != nil {
                log.Println("Cannot read file:", err)
        }
        defer file.Close()
        img, _, err := image.Decode(file)
        if err != nil {
                log.Println("Cannot decode file:", err)
        }
        size := img.Bounds().Size()
        for i := 0; i < size.X; i++ {
                var y []color.Color
```

```
                for j := 0; j < size.Y; j++ {
                        y = append(y, img.At(i, j))
                }
                grid = append(grid, y)
        }
        return
}

// save the image to file
func save(filePath string, grid [][]color.Color) {
        xlen, ylen := len(grid), len(grid[0])
        rect := image.Rect(0, 0, xlen, ylen)
        img := image.NewNRGBA(rect)
        for x := 0; x < xlen; x++ {
                for y := 0; y < ylen; y++ {
                        img.Set(x, y, grid[x][y])
                }
        }
        file, err := os.Create(filePath)
        if err != nil {
                log.Println("Cannot create file:", err)
        }
        defer file.Close()
        png.Encode(file, img.SubImage(img.Rect))
}
```

Now that you have the code, you can profile it. There are two ways of profiling your code using `pprof`:

- Use the `go test` tool and the flag `-cpuprofile` on a benchmark function that has the code you want to profile.
- Add code from the `runtime/pprof` package into the code you want to profile, then run the code.

Using the `go test` tool with the `-cpuprofile` flag is relatively easier, so take a look at that first:

```
func BenchmarkResize(b *testing.B) {
        for i := 0; i < b.N; i++ {
                grid := load("monalisa.png")
                resized := resize(grid, 3.0)
                save("resized.png", resized)
        }
}
```

As you can see, the benchmark function is just like any other benchmark function. In this function, first, you load the PNG file and create a grid, then you run the `resize` function on the grid, and finally save the resized grid to another file.

Start profiling like this:

```
% go test -cpuprofile cpu.prof -bench=Resize -run=XXX
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch19_benchmarking
BenchmarkResize-10             6         181341944 ns/op
PASS
ok      github.com/sausheong/gocookbook/ch19_benchmarking       1.928s
```

Use the `-cpuprofile` flag and pass in the filename `cpu.prof` to create the profile into this file. The code takes some time, and the benchmark function only runs six times in a second. You'll see a *cpu.prof* file being created. It's a binary file, so you can't open it up to look inside. Instead, you'll have to use `pprof` to analyze the profile.

The second way to create profiles is to add profiling code directly into the program code using the `runtime/pprof` package. There's quite a bit of profiling code to write, but fortunately, there is a small package that allows you to profile a program a lot easier:

```
func main() {
        defer profile.Start(profile.ProfilePath(".")).Stop()
        grid := load("monalisa.png")
        resized := resize(grid, 3.0)
        save("resized.png", resized)
}
```

Take the same few lines of code you wanted to profile earlier and put them into a `main` function that is in a `main` package. You need to use the `github.com/pkg/profile` package and add a simple line of code to start profiling and create a CPU profile.

Then create the binary executable file:

```
% go build -o resize
```

This will create the `resize` program. When you run this program, it will generate a CPU profile in a file named *cpu.pprof*.

Now that you have the CPU profiles—one created using `go test` and the other by running profiling code in a program—you can analyze them using `pprof`. You'll use the same method of analyzing the two CPU profiles generated by the two different methods, but you will quickly realize the profiles are different since they are created differently.

The `pprof` tool can be accessed in a number of ways, but the easiest way to visualize is probably using the web interface. Before you can use the `pprof` web interface, you need to install Graphviz. On a macOS machine, install it using Homebrew:

```
% brew install graphviz
```

For Windows machines, download the appropriate package from Graphviz and install it by running the installer.

For Linux machines, use your usual package manager to install Graphviz. For example, for Debian machines:

```
% sudo apt install graphviz
```

Once you have done that, you can start the web interface for the `pprof` tool like this:

```
% go tool pprof -http localhost:8080 cpu.prof
Serving web UI on http://localhost:8080
```

Notice that you asked for it to be started at port 8080 and using the *cpu.prof* file. Running this command on the command line will also pop up the browser. Figure 19-1 shows the screenshot of the web UI started by the `pprof` tool, showing the CPU profile generated earlier.

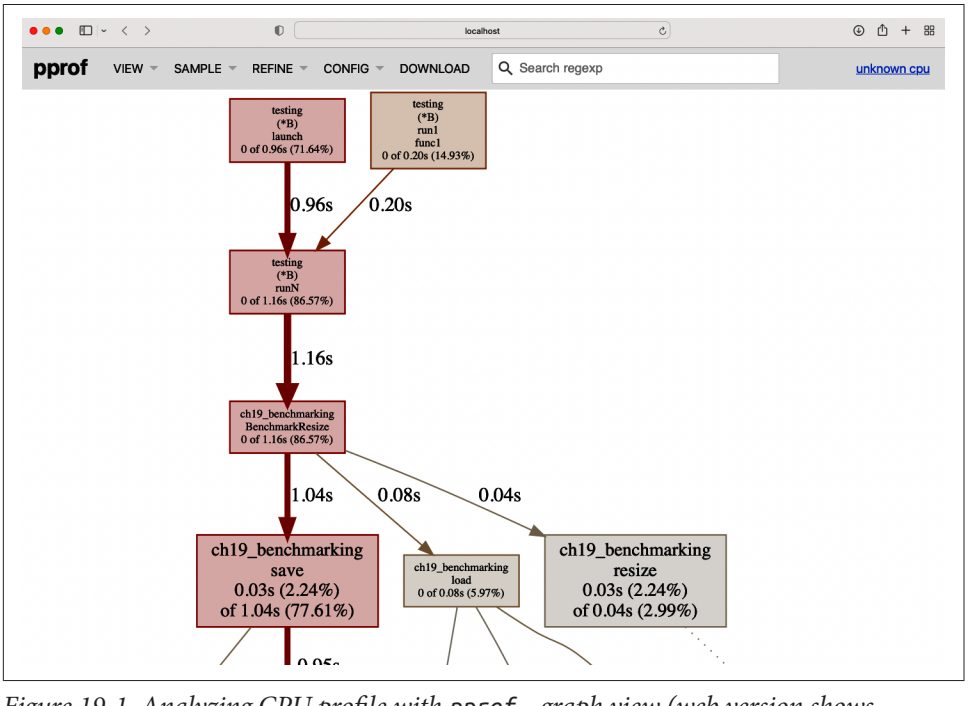

Figure 19-1. Analyzing CPU profile with `pprof`—graph view (web version shows different shades of red; print version shows grayscale); darker colors indicate that more time is spent on those tasks

You can see from the web UI that `testing.B` takes up 86% of the CPU time. Within `BenchmarkResize`, 77% of the time is spent on `save`, 6% is spent on `load`, and only 3% is spent on `resize` function. Within the `save` function, most of the time is spent on encoding the image. Within the `load` function, most of the time is spent on decoding the image.

The web interface can do multiple views. The default view is the graph view, but you can also choose the flame graph view by selecting the VIEW menu and then choosing Flame Graph from the list. Figure 19-2 shows the flame chart view of the web UI started by the `pprof` tool.

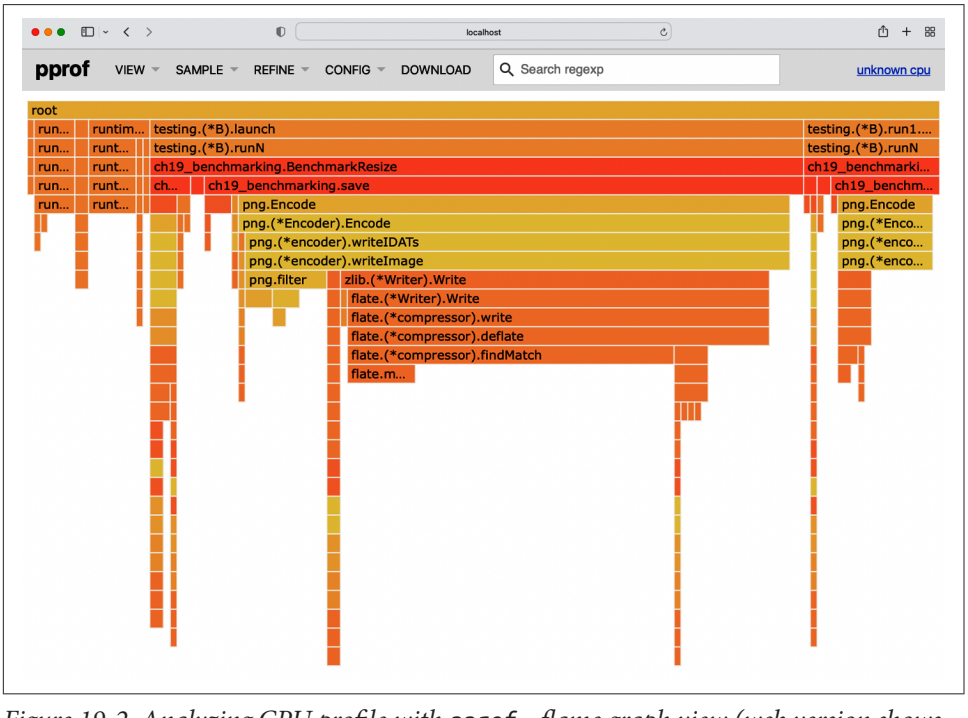

Figure 19-2. Analyzing CPU profile with `pprof`—flame graph view (web version shows red, orange, and yellow; print version shows grayscale); darker colors indicate that more time is spent on those tasks

This provides another perspective. Click the horizontal bars, and they will be expanded to show more. It is interesting that `BenchmarkResize` is shown more than once on the flame graph. Remember that you actually ran the loop six times. The profiling manages to capture a bit more than one of the iterations.

What happens if you tell the benchmark function to iterate only once?

```
% go test -cpuprofile cpu.prof -bench=Resize -run=XXX -benchtime=1x
goos: darwin
goarch: arm64
pkg: github.com/sausheong/gocookbook/ch19_benchmarking
BenchmarkResize-10             1         196764042 ns/op
PASS
ok      github.com/sausheong/gocookbook/ch19_benchmarking       0.497s
```

As you can see, there is only one `BenchmarkResize`. Figure 19-3 shows the flame chart when running only one iteration in the benchmark function.

Figure 19-3. Flame graph when running only one iteration in the benchmark function (web version shows red, orange, and yellow; print version shows grayscale); darker colors indicate that more time is spent on those tasks

Another interesting view is the source view, which shows the flat and cumulative timings for the significant lines of code. There could be a lot of code shown, so you can narrow it down to the functions you are interested in by searching through the code. Figure 19-4 shows the source view of the web UI started by the `pprof` tool.

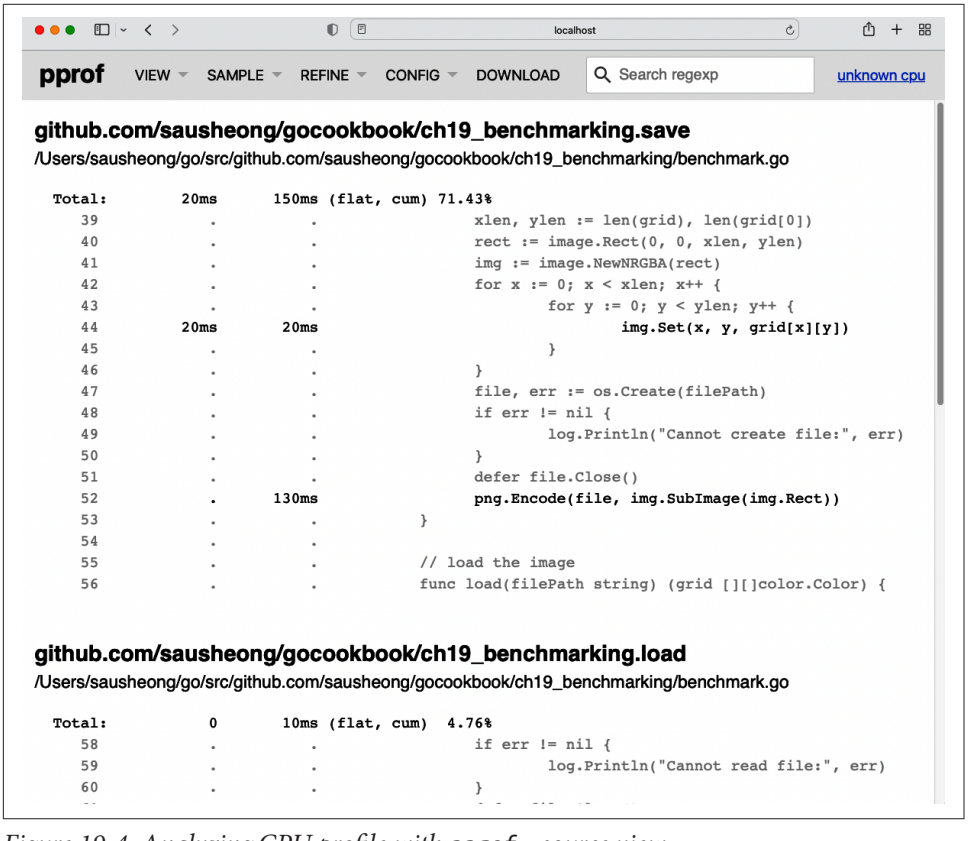

Figure 19-4. Analyzing CPU profile with `pprof`—source view

You've seen the profile created using the benchmark function. Take a quick look at the profile created by placing the profiling code in the program.

Figure 19-5 shows how you can analyze the CPU profile by adding profiling code.

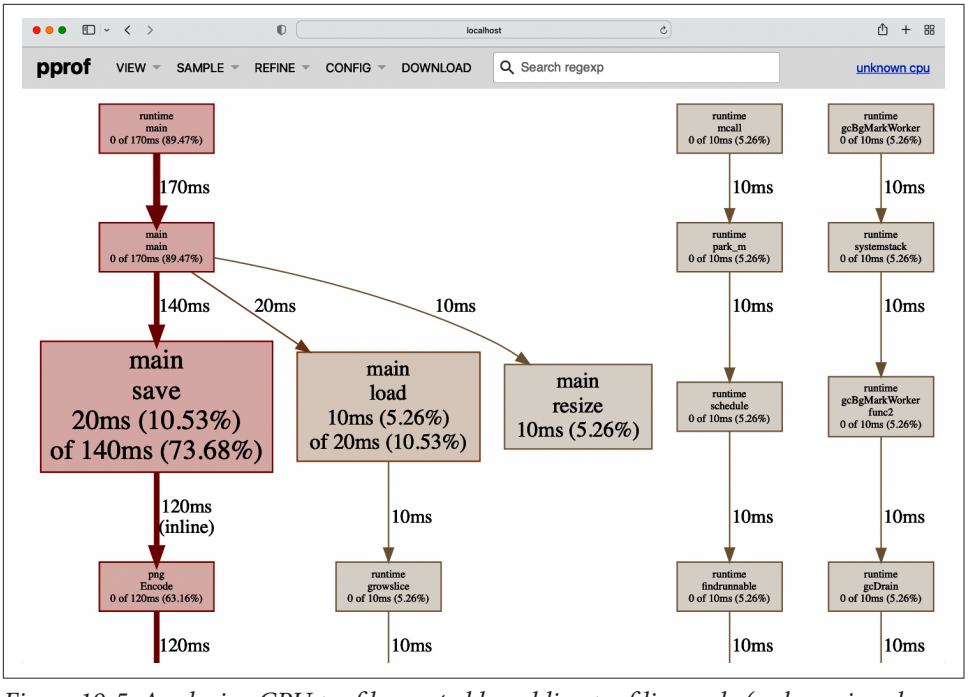

Figure 19-5. Analyzing CPU profile created by adding profiling code (web version shows different shades of red; print version shows grayscale); darker colors indicate that more time is spent on those tasks

You can see that the benchmarking parts are now gone. The percentages differ slightly, but they should be the same.

Index

Symbols

A

B

C

D

E

F

G

H

Q

R

S

T

U

V

W

About the Author

Sau Sheong Chang is the Deputy Chief Executive of the Government Technology Agency of Singapore, more commonly known as GovTech. GovTech is the government agency responsible for delivering smart city and digital services to the Singapore government as well as the public.

Sau Sheong has been in the software development industry for more than 28 years and has been involved in building software products in many industries and using various technologies. He is an active member of various software development communities, previously Java and Ruby, but now focuses mostly on the Go community. He runs meetups and gives talks at conferences all around the world on Go and also on topics related to his work, especially on sustainability, smart cities, government, and AI. He also runs GopherCon Singapore, one of the largest community-led developer conferences in Southeast Asia, and has been doing so since 2017. Sau Sheong has written four programming books, three in Ruby and the last one in Go.

Colophon

The animal on the cover of *Go Cookbook* is a stoat (*Mustela erminea*). Stoats are a species of mustelid (carnivorous mammal) related to weasels and otters. They can be found across Eurasia and North America in grasslands, farmlands, orchards, woodlands, heathland, and moorland.

Stoats are small mammals with long, lithe bodies that can grow up to 25 centimeters long. Most of their fur is brown with the exception of their white underside and black tail tips. Stoats also molt in the winter. In colder climates, their fur coat becomes entirely white. In warmer climes, their fur may stay brown or have a patchy appearance.

Stoats tend to hunt during the day and can eat up to 25% of their own body weight. Their slim shape makes them speedy, agile hunters. Their diet consists of other small mammals, such as rabbits and water voles.

The population of stoats can vary based on availability of food. For example, if rabbits are plentiful, there will be a plentiful number of stoats. However, if the rabbit population declines, so does the stoat population. Overall, the stoat population is not at risk of endangerment and is categorized as least concern on endangered species lists. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from *Shaw's Zoology*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.