GO FUNDAMENTALS

MARK BATES | CORY LANOU

GOPHER GUIDES

Foreword by STEVE FRANCIA

FREE SAMPLE CHAPTER

Go Fundamentals

This page intentionally left blank

Go Fundamentals

Mark Bates
Cory LaNou

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Cover image: T-flex/Shutterstock

Figures 7.1, 7.2: Microsoft

Library of Congress Control Number: 2022943902

ISBN-13: 978-0-13-791830-0
ISBN-10: 0-13-791830-5

ScoutAutomatedPrintCode

Pearson's Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content accurately reflects the histories and experiences of the learners we serve.
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview).

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

Please contact us with concerns about any potential bias at https://www.pearson.com/report-bias.html.

This page intentionally left blank

For Rachel, Dylan, Leo, and Ringo.
—Mark Bates

For Karie, Logan, and Megan.
—Cory LaNou

Contents

Foreword

I feel excited, honored, and a bit shocked to be writing these words.

I've known Mark and Cory for more than a decade, and as the Go project has evolved beyond our 1.0 vision, I've really been hoping for someone to write a follow up to the excellent Go 1.0 books. I'm really grateful for the good fortune I've had to be so deeply involved in the Go project and community. What began with me typing the first few lines of code into Hugo,[1] Cobra,[2] and Viper[3] over 10 years ago has exploded in ways I couldn't have dreamed of. It's been the privilege of a lifetime to lead the Go project alongside my partners Russ Cox and Sameer Ajmani and to work alongside the luminary programmers both on the Go team at Google and across the Go community.

I met Mark when we both spoke at the first Gotham Go.[4] I was instantly drawn away from him. I found his on-stage persona to be a bit too much. Later that year, the organizers of GopherCon[5] asked me to lead the lightning talk program and said they had the perfect partner for me. You guessed it—Mark. I learned that Mark's stage persona was indeed who he was off stage. I also learned that he was a loyal friend who would do anything for the Go community, whose humor and courage was boundless… limitless… maybe a bit too much. Mark and I have paired on many stages and projects over the past 10 years, and Mark injected a spirit of excitement into everything he did. He's become a dear friend, and I'm grateful to have been on so many adventures with him.

I met Cory at the second GopherCon and instantly was drawn to him. Cory is a natural teacher—captivating and empathetic. He cares deeply about the Go community and ensuring that Go is accessible to all, especially the folks new to Go. I've worked with him on a variety of community efforts, and I've always come away impressed with the depth he shows in his knowledge about Go and the learning experience.

Mark and Cory have been working together for many years now as the dominant training duo for Go under the name Gopher Guides. Together they have produced excellent training programs for clients that include many notable brands found on the Fortune 500. They have expertise in both Go and in empathic learning, forged over thousands of hours of classroom-style instruction. They are the perfect pair to author what is destined to become "The Go Book" for Go's second phase (Go with Modules & Generics).

This book leans on Mark and Cory's years of experience in the field and in the Go community to take a very grassroots approach to learning, and it is the guide for

---

1. https://gohugo.io/
2. https://pkg.go.dev/github.com/spf13/cobra
3. https://pkg.go.dev/github.com/spf13/viper
4. https://gothamgo.com/
5. https://www.gophercon.com/

programmers to become gophers. It also leverages their practical experience writing Go libraries and applications to present to the reader pragmatic solutions and simple explanations.

This book guides you like an old friend would, telling you the technical approaches to things but also relaying the cultural norms and idioms. It answers questions that the majority of books don't even think of because Mark and Cory have heard these questions asked by real people in a classroom so many times. Through reading and applying what is in this book, more than any other of its kind, you will progress from becoming a programmer to being a programmer who writes Go to being a gopher.

I am excited for the journey you are about to take with this book. I'm sure that you will sense the same excitement and empathy through these pages that I have experienced through all the adventures I've had with Mark and Cory. My hope for you is that as you learn Go, you will, as I did when I first discovered Go over a decade ago, fall in love with programming again.

—*Steve Francia*
@spf13

Preface

Over the years, we have been fortunate to have trained thousands of developers in Go. It has been an incredible experience for us. We are developers who care about developers, so to be able to help developers grow is a great honor for us. This book is a culmination of that experience.

With nearly a half-century of experience between us, we both love Go. Go is fast, efficient, and fairly straightforward. We both believe that Go is a great first language to learn.

Unlike previous languages we have both used, there is relatively little "magic" in the language. Developers are rarely left scratching their heads as to where a function or type came from. Although the language does offer tools, such as the `reflect`[1] package, those tools are not used often. As part of this philosophy, the general consensus in the community is to favor using the standard library over third-party libraries. Again, this stems from experience with languages where bloated dependencies caused all sorts of problems. Also, early in the language's history, there was no package manager, so managing dependencies was a challenge.

The Go compiler is another reason why Go has become so popular. The speed of the compiler is a huge factor in the development of Go. Fast compilation times mean a faster feedback cycle for developers. Large Go applications can be compiled in seconds. Small Go applications, when using `go run`, compile and execute so quickly that it makes Go feel like a scripting language. This fast compilation and execution time extends to the running of tests as well. It is not uncommon for a Go developer to run the entire test suite every time they save a file. This is not something that can be done with other languages.

The Go compiler, the language's type system, and its fast compiler mean that a lot of common bugs are very quickly caught. This is a great advantage for Go developers. We are able to focus on business logic, knowing that the compiler will catch any stupid mistakes we might make, such as using a variable before it is defined or forgetting to use the `range` keyword in a loop.

For example, in Listing P.1, we are doing just that. You can see from the output that the compiler caught the error and did so very quickly.

1. https://pkg.go.dev/reflect

Listing P.1 Quick Feedback Cycle Thanks to the Go Compiler

```
package main

import (
    "fmt"
    "os"
    "os/exec"
    "time"
)

func main() {
    letters := []string{"a", "b", "c"}

    for _, l := letters {
        fmt.Println(l)
    }
}
```

```
$ go run .

# demo
./main.go:13:11: syntax error: cannot use _, l := letters as value
```

```
Duration:    64.341084ms
Go Version: go1.19
```

We discuss more about looping in Chapter 3 and more about how the Go compiler and runtime work as we progress throughout the book.

Finally, Go has a great concurrency story. The language was built from the ground with concurrency in mind. This means you don't need to worry about threading, spawning processes, or anything like that. You can write concurrent code quickly and easily.

In the code snippet in Listing P.2, we are using goroutines, channels, and synchronization types, such as `sync.WaitGroup`[2] to create an application that properly handles concurrency.

Listing P.2 Go Concurrency in Action

```
func main() {
    // create a channel of type int
    ch := make(chan int)
```

2. https://pkg.go.dev/sync#WaitGroup

```
    // create a wait group to track
    // the goroutines
    var wg sync.WaitGroup

    // create some goroutines to
    // listen for messages on the
    // channel
    for i := 0; i < 4; i++ {

        // increment the wait group
        wg.Add(1)

        // create a goroutine
        // to call the sayHello function
        go func(id int) {

            // call the sayHello function
            sayHello(id, ch)

            // decrement the wait group
            // when the sayHello function
            // exits
            wg.Done()
        }(i + 1)
    }

    // send messages on the channel
    for i := 0; i < 10; i++ {
        ch <- i
    }

    // close the channel to signal
    // the goroutines to exit
    close(ch)

    // wait for all goroutines to finish
    wg.Wait()
}

func sayHello(id int, ch chan int) {

    // listen for messages on the channel
    // loop exits when the channel is closed
    for i := range ch {
```

```go
        fmt.Printf("Hello %d from goroutine %d\n", i, id)

        // simulate a long-running task
        sleep()
    }
}
```

```
$ go run .



Hello 2 from goroutine 4
Hello 1 from goroutine 2
Hello 3 from goroutine 3
Hello 0 from goroutine 1
Hello 4 from goroutine 1
Hello 5 from goroutine 2
Hello 6 from goroutine 1
Hello 7 from goroutine 1
Hello 8 from goroutine 4
Hello 9 from goroutine 3
```

```
Go Version: go1.19
```

You can read more about Go's concurrency model in Chapters, 11, 12, and 13.

How to Read This Book

We have laid this book out with the idea that it is meant to be read from start to finish. Each chapter, and each example, build on previous examples. While those with some experience with Go might be tempted to jump straight to later chapters, we really recommend reading the whole book.

We have gone through great effort to try to make sure that examples and chapters don't contain any new information that we haven't covered up to that point. In Chapter 9, for example, we cover errors in Go. To understand how errors work, you first need to learn about interfaces, which we cover in Chapter 8. Occasionally, we do have to break this rule, though. In those instances, we try our best to explain the concept or point to the chapter where it is explained.

About the Examples

As mentioned, we spent considerable time designing the examples in this book and our training materials to be as clear and concise as possible. We also wanted to make sure that all of the code, output, and documentation were accurate.

To ensure that all of these things are true, we used a system designed give us this accuracy. Knowing that every code snippet is from a real file, every command is actually executed, every output is exact, and every piece of documentation is up to date means we can be confident in the materials we present.

In Listing P.3, you see an example of what one of these documents looks like.

Listing P.3 Sample of How the Book Was Written

```
### Commands and Documentation

In <ref>exit</ref>, we see a snippet of code.

This will execute the command 'go run .' in the 'src/bad' directory.
This command is expected to exit with a code of '1'.
The output of the command is automatically captured and inserted into
➥the document.

<figure id="exit" type="listing">
<code src="module.md#exit" esc></code>
<figcaption>Handling a non-successful exit code.</figcaption>
</figure>

In <ref>panic</ref>, we can see the output of the non-successful program.
➥Notice, at the top of the output we see the command used that was executed,
➥'$ go run .'. At the bottom we can see that the program was executed with,
➥Go version '1.19'.

<figure id="panic">
<go src="src/bad" run="." exit="1"></go>
<figcaption>Output of a non-successful program.</figcaption>
</figure>

While this book was written with Go version '1.18', all of the final
➥examples, and documentation, were run with Go version '1.19'.

In <ref>appendf.doc</ref>, we can see an example of documentation that is
➥being inserted into the document. In this case, we are inserting the
➥documentation for the new Go '1.19' function, 'fmt.Appendf'. Again, just
➥like the command output in <ref>panic</ref>, the command used to get the
➥documentation and the Go version is shown at the bottom of the output.
```

(*continued*)

```
<figure id="appendf.doc">
<go doc="fmt.Appendf"></go>
<figcaption>The 'fmt.Appendf' function documentation.</figcaption>
</figure>
```

The output of Listing P.3 is included in the next section.

Commands and Documentation

Listing P.4 shows a snippet of code.

It executes the command `go run .` in the `src/bad` directory. This command is expected to exit with a code of `1`. The output of the command is automatically captured and inserted into the document.

Listing P.4 Handling a Nonsuccessful Exit Code

```
<go src="src/bad" run="." exit="1"></go>
```

In Listing P.5, you can see the output of the nonsuccessful program. Notice that the top of the output, you see the command used that was executed: `$ go run .` At the bottom, you can see that the program was executed with Go version `1.19`.

Listing P.5 Output of a Nonsuccessful Program

```
$ go run .

panic: Hello, World!

goroutine 1 [running]:
main.main()
        ./main.go:5 +0x2c
exit status 2

--------------------------------------------------------------------------------
Go Version: go1.19
```

While this book was written with Go version `1.18`, all of the final examples and documentation were run with Go version `1.19`.

In Listing P.6, you can see an example of documentation that is being inserted into the document. In this case, we are inserting the documentation for the new Go `1.19` function, `fmt.Appendf`. Again, just like the command output in Listing P.5, the command used to get the documentation, `$ go do fmt.Appendf`, is shown at the top of the output, and the Go version is shown at the bottom of the output.

Listing P.6 The `fmt.Appendf` Function Documentation

```
$ go doc fmt.Appendf

package fmt // import "fmt"

func Appendf(b []byte, format string, a ...any) []byte
    Appendf formats according to a format specifier, appends the result to the
    ➥byte slice, and returns the updated slice.
---------------------------------------------------------------------------------
Go Version: go1.19
```

Summary

Contained within these pages are the concepts, types, packages, idioms, and other features of Go that we believe to be the most fundamental to understanding the language.

Go, its ecosystem, and its community are large, complex, and vibrant. Although we wish we could have included more in this book, it would've been impossible to print, and we would have never finished it.

Our goal, by the end of this book, is to help you be a knowledgeable Go programmer—one who not only feels confident using the language but understands how to write idiomatic Go code, tests, and is a good community member.

Finally, dear reader, thank you for your support. We consider it a privilege to be able to share this book and our knowledge with you. We love Go, and we hope you do, too.

—*Mark Bates and Cory LaNou*
August 2022

Register your copy of *Go Fundamentals* on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780137918300) and click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access any available bonus materials. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

This page intentionally left blank

Acknowledgments

Authoring a book is incredibly hard work, whether you are self-publishing or working with a great publisher like Addison-Wesley. It requires long hours of writing, creating just the right sample code, and refactoring. This is usually where the self-publishing author stops. When working with a publisher, however, this only the start of a new phase in the process.

After a book is written and submitted, a team of people jump in to try and make the book the best it can be. There are technical reviewers who are there to make sure the code and the discussion of the code are correct. There are copy editors who drill into the minutiae of grammar, spelling, punctuation, and other stylistic issues. There is the production team who is responsible for the final delivery of the book. There is the cover designer, the proofreader, the indexer, and more. As the adage says, "It takes a village."

Gopher Guides would like to thank Kim Spenceley, our editor, and Debra Williams-Cauley, our managing editor, for all of their incredible support since we first started this journey. Their optimism and excitement has kept us going through this process. Without Kim and Debra, there would be no book. Thank you two so very much.

In addition to Kim and Debra, Gopher Guides would like to the following people at Addison-Wesley for their help in the process of making this book a reality:

Copy editor and project manager: Charlotte Kughen

Production manager: Sandra Schroeder

Production editor: Julie Nahil

Cover designer: Chuti Prasertsith

Development editor: Chris Zahn

Indexer: Cheryl Lenser

Proofreader: Sarah Kearns

The rest of the village at Addison-Wesley

The Go community has been a huge supporter of Gopher Guides over the years. We would like to thank everyone on the Go team for their dedication and hard work in creating a language that is loved by so many. Thank you to all those who have committed to the Go project over the years. Again, it takes a village.

In particular, Gopher guides would like to thank the following Go community members who have helped us along the way.

Steve Francia

Ashley Willis (McNamara) (who designed the Gopher Guides logo)

Mat Ryer

Carmen Andoh

Johnny Boursiquot

Tim Raymond

Antonio Pagano

Renee French (for creating the Gopher logo)

Ron Evans

Bill Kennedy

Brian Ketelsen

Bryan Liles

Ben Johnson

Dave Cheney

Matt Aimonetti

Francesc Campoy

From Mark Bates

In addition to everyone that Cory and I have thanked already, I would like to make a few acknowledgments of my own.

First, I would like to thank my business partner and friend, Cory LaNou. Cory and I knew each other well in the community, but we really got to know each other and became good friends on a trip to GopherCon India in Bengaluru (Bangalore), India, and in Dubai, United Arab Emirates. We both spoke at the conferences, which were great, but it was the extra curriculum that truly bonded us. We rode camels, drove dune buggies, and went to a desert oasis. It was a great experience.

About a year later, in 2017, Cory and I decided that we wanted to start a company to help people learn Go. With that, Gopher Guides was born. I have never worked anywhere for longer than two and a half years. Gopher Guides, at the time of writing, has been going for more than five years. That is more than double the time I've worked at any job. The reason for this is simple: Cory LaNou is an excellent partner and a great friend. I have never felt as supported and as understood while working at any other company as I do working with Cory.

Cory, thank you for your patience, understanding, and support. I couldn't be prouder of what we have built together—this great company and our enduring friendship.

Next, I want to thank Kim Spenceley and Debra Williams-Cauley. I have known both of them for a long time. Debra was the editor on my first two books for Addison-Wesley in 2009 and 2012. Kim was the publishing coordinator of the first book. Both of these strong women have been of great support to me over the years. I always feel lucky that I was able to find a publishing company like Addison-Wesley—with editors like Debra and Kim—that is so focused on author success. Thank you both for helping me to first realize my dream of having a book published, and then for encouraging me to continue writing.

Finally, I would like to thank my family. There is no better partner, friend, and supporter than my wife, Rachel. She is an actual wonder woman. The saying "behind every good man is a great woman" is about her. She is a ball of energy, enthusiasm, and spirit. She has a goal of running a half marathon in every U.S. state, and she's more than halfway there. I get winded driving 13.1 miles. She is the morning person to my night owl. Rachel is a successful business woman who has spent her career being an advocate for women in the workplace. She is the mentor that she never had to a generation of women in business. I know several of these women, and to hear them talk about Rachel with such respect and appreciation is heart-warming.

Lastly, in addition to all of her support, love, and dedication over the years, Rachel has given me the greatest gift of all; our two sons, Dylan and Leo, are wonderful people. They are kind, caring, and honest. They are both very smart, talented, and hilarious people who continue to amaze me and make me glow with pride. Each night, we eat dinner together and then settle in, as a family, to watch TV. Our dog Ringo curls up at the end of the sofa, and we all enjoy each other's company. It's at those moments I feel truly thankful and happy to be surrounded by their love.

Thank you Rachel, Dylan, and Leo (and Ringo—the dog, not the Beatle—though he's cool, too!) for all of your support and love. I love you all to the very depths of my heart.

From Cory LaNou

First, and most obvious, my thanks go to my business partner, Mark Bates. Mark and I met through the Go community and have become not just great partners in our business but also great friends. We both share a passion for Go and training, which is why I believe we have been so successful together.

A big thanks to Levi Cook for introducing me to the Go language in March 2012 and encouraging me to start one of the first Go meetups in the world in Denver, Colorado. Levi not only introduced me to Go but also to open source and, most important, VIM!

I also want to recognize some of the brightest and most talented developers I have ever worked with on the InfluxDB project. Ben Johnson for having an ability to name the impossible things. Jason Wilder for opening my mind to taking the most complex of issues and breaking them into their smallest components. Philip O'Toole for challenging every new concept I introduced to the Go code base. David Norton for being the nicest person I've ever worked with (and giving the best SQL presentation ever!). Joe LeGasse for never hesitating to pair program with me. And all the other amazing people I worked with on the InfluxDB team.

When I moved back to Wisconsin to be around all my family, I knew I would miss Denver's rich technology community. Thankfully, Doug Rhoten was heading up the Chippewa Valley Developers meetup and made me feel incredibly welcome. Thank you, Doug, for working so hard for the local technology community!

And lastly, but certainly not least, my family! My wife, Karie, has been nothing but supportive, even when I have done the craziest of business ventures. Without her support and belief in me, I would not be where I am today. My son, Logan, and daughter, Megan, who thank me for all the long hours I put in so that they can have bacon for breakfast!

This page intentionally left blank

About the Authors

Mark Bates is cofounder and instructor at Gopher Guides, the industry leader for Go training, consulting, and conference workshops. After graduating with a degree in music from the Liverpool Institute for Performing Arts in 1999, Mark joined the original dot-com boom as a software engineer and began his career as a technologist. Since then, Mark has been fortunate enough to work with some of the world's largest and most innovative companies, including Apple, Uber, and Visa.

When he first discovered Go in the summer of 2013, Mark was immediately drawn to the language and its ecosystem. In 2014, Mark attended the first GopherCon, where he met Cory LaNou. For seven years, Mark hosted the GopherCon lightning talks and has been proud to introduce hundreds of new speakers to the community. Mark has spoken at conferences around the world and is a regular on the *Go Time* podcast.

When not coding or writing about coding, Mark enjoys spending time with his family, traveling, and recording music. Mark is currently completing a master's degree in music production from Berklee College of Music in Boston, Massachusetts.

Cory LaNou is a full stack technologist who has specialized in start-ups for the last 20 years. Cory has started several technology companies over the years, including companies such as Pulsity, a web consulting firm that operated in the late 1990s. From that company, Local Launch (an Internet marketing technology company) was formed and later sold to RH Donnelly in 2006.

Cory has deep experience in the Go world, having worked on the Core engineering team that contributes to InfluxDB, which is a highly scalable, distributed time series database written in Go. Cory has worked on several other Go projects with a focus on profiling and optimizing applications, using advanced profiling concepts such as torch graphs and tracing profiles.

Cory has deep ties to the Go community and started Denver Gophers, one of the very first Go meetups in the world, and he assisted in the very first GopherCon. Cory has created and led numerous Go workshops and training courses and has several published online articles related to Go. He continues to help organize and lead several community technology meetups and mentors new developers. Cory is also a partner at Gopher Guides, the industry leader for Go training and conferences.

When Cory isn't working on writing new Go training material, he volunteers his time to help local entrepreneurs start up their businesses by offering free services and business advice—such as how to incorporate their business and get their web presence set up—and giving them the moral support they need to make their dreams a reality.

This page intentionally left blank

13

Synchronization

In Chapter 11, we explained how to use channels for passing data between goroutines. Then in Chapter 12, we discussed how to use the `context`[1] package to manage the cancellation of goroutines. In this chapter, we cover the final part of concurrent programming: synchronization.

We show you how to wait for a number of goroutines to finish. We explain race conditions,[2] how to find them using Go's `-race`[3] flag, and how to fix them with `sync.Mutex`[4] and `sync.RWMutex`.[5]

Finally, we discuss how to use `sync.Once` to ensure a function is only executed one time.

Waiting for Goroutines with a WaitGroup

Often, you might want to wait for a number of goroutines to finish before you continue your program. For example, you might want to spawn a number of goroutines to create a number of thumbnails of different sizes and wait for them all to complete before you continue.

The Problem

Consider Listing 13.1. We launch `5` new goroutines, each of which creates a thumbnail of a different size. We then wait for all of them to complete.

Listing 13.1 Launching Multiple Goroutines to Complete One Task

```go
func Test_ThumbnailGenerator(t *testing.T) {
    t.Parallel()

    // image that we need thumbnails for
    const image = "foo.png"
```

(continued)

1. https://pkg.go.dev/context
2. https://en.wikipedia.org/wiki/Race_condition
3. https://golang.org/doc/articles/race_detector
4. https://pkg.go.dev/sync#Mutex
5. https://pkg.go.dev/sync#RWMutex

```
    // start 5 goroutines to generate thumbnails
    for i := 0; i < 5; i++ {

        // start a new goroutine for each thumbnail
        go generateThumbnail(image, i+1)

    }

    fmt.Println("Waiting for thumbnails to be generated")
}
```

The `generateThumbnail` function, Listing 13.2, generates a thumbnail of the specified size. In this example, we sleep one millisecond per "size" of thumbnail to simulate the time it takes to generate the thumbnail. For example, if we call `generateThumbnail("foo.png", 200)`, we sleep 200 milliseconds before returning.

Listing 13.2 A Test Exiting before All Goroutines Have Finished

```
func generateThumbnail(image string, size int) {

    // thumbnail to be generated
    thumb := fmt.Sprintf("%s@%dx.png", image, size)

    fmt.Println("Generating thumbnail:", thumb)

    // wait for the thumbnail to be ready
    time.Sleep(time.Millisecond * time.Duration(size))

    fmt.Println("Finished generating thumbnail:", thumb)
}
```

```
$ go test -v

=== RUN   Test_ThumbnailGenerator
=== PAUSE Test_ThumbnailGenerator
=== CONT  Test_ThumbnailGenerator
Waiting for thumbnails to be generated
--- PASS: Test_ThumbnailGenerator (0.00s)
PASS
ok      demo    0.408s
```

```
Go Version: go1.19
```

As you can see from the test output in Listing 13.2, the test exits before the thumbnails are generated.

Our tests exit prematurely because we have not provided any mechanics to ensure that we wait for all of the thumbnail goroutines to finish before we continue.

Using a WaitGroup

To help us solve this problem, we can use a `sync.WaitGroup`,[6] Listing 13.3, to track how many goroutines are still running and notify us when they have all finished.

Listing 13.3 The `sync.WaitGroup` Type

```
$ go doc -short sync.WaitGroup

type WaitGroup struct {
        // Has unexported fields.
}
    A WaitGroup waits for a collection of goroutines to finish. The main
    ➥goroutine calls Add to set the number of goroutines to wait for. Then
    ➥each of the goroutines runs and calls Done when finished. At the same
    ➥time, Wait can be used to block until all goroutines have finished.

    A WaitGroup must not be copied after first use.

func (wg *WaitGroup) Add(delta int)
func (wg *WaitGroup) Done()
func (wg *WaitGroup) Wait()

Go Version: go1.19
```

The principle is simple: We create a `sync.WaitGroup` and use the `sync.WaitGroup.Add`[7] method to add to the `sync.WaitGroup` for each goroutine we want to wait for. When we want to wait for all of the goroutines to finish, we call the `sync.WaitGroup.Wait`[8] method. When a goroutine finishes, it calls the `sync.WaitGroup.Done`[9] method to indicate that the goroutine is finished.

The `Wait` Method

As the name suggests, a `sync.WaitGroup` is about waiting for a group of tasks, or goroutines, to finish. To do this, we need a way of blocking until all of the tasks have finished. The `sync.WaitGroup.Wait` method in Listing 13.4 does exactly that.

6. https://pkg.go.dev/sync#WaitGroup
7. https://pkg.go.dev/sync#WaitGroup.Add
8. https://pkg.go.dev/sync#WaitGroup.Wait
9. https://pkg.go.dev/sync#WaitGroup.Done

The `sync.WaitGroup.Wait` method blocks until its internal counter is zero. When the counter is zero, it means that all of the tasks have finished, and we can unblock and continue.

Listing 13.4 The `sync.WaitGroup.Wait` Method

```
$ go doc sync.WaitGroup.Wait

package sync // import "sync"

func (wg *WaitGroup) Wait()
    Wait blocks until the WaitGroup counter is zero.

--------------------------------------------------
Go Version: go1.19
```

The Add Method

For a `sync.WaitGroup` to know how many goroutines it needs to wait for, we need to add them to the `sync.WaitGroup` using the `sync.WaitGroup.Add` method, Listing 13.5.

Listing 13.5 The `sync.WaitGroup.Add` Method

```
$ go doc sync.WaitGroup.Add

package sync // import "sync"

func (wg *WaitGroup) Add(delta int)
    Add adds delta, which may be negative, to the WaitGroup counter.
    ➥If the counter becomes zero, all goroutines blocked on Wait are released.
    ➥If the counter goes negative, Add panics.

    Note that calls with a positive delta that occur when the counter is
    ➥zero must happen before a Wait. Calls with a negative delta, or calls
    ➥with a positive delta that start when the counter is greater than zero,
    ➥may happen at any time. Typically this means the calls to Add should
    ➥execute before the statement creating the goroutine or other event to be
    ➥waited for. If a WaitGroup is reused to wait for several independent
    ➥sets of events, new Add calls must happen after all previous Wait calls
    ➥have returned. See the WaitGroup example.

--------------------------------------------------
Go Version: go1.19
```

The `sync.WaitGroup.Add` method takes a single integer argument, which is the number of goroutines to wait for. There are, however, some caveats to be aware of.

Adding a Positive Number

The sync.WaitGroup.Add method accepts an int argument, which is the number of goroutines to wait for. If we pass a positive number, the sync.WaitGroup.Add method adds that number of goroutines to the sync.WaitGroup.

As you can see from the test output in Listing 13.6, the sync.WaitGroup.Wait method blocks until the internal counter of the sync.WaitGroup reaches zero.

Listing 13.6 Adding a Positive Number of Goroutines

```
func Test_WaitGroup_Add_Positive(t *testing.T) {
    t.Parallel()

    var completed bool

    // create a new waitgroup (count: 0)
    var wg sync.WaitGroup

    // add one to the waitgroup (count: 1)
    wg.Add(1)

    // launch a goroutine to call the Done() method
    go func(wg *sync.WaitGroup) {

        // sleep for a bit
        time.Sleep(time.Millisecond * 10)

        fmt.Println("done with waitgroup")

        completed = true

        // call the Done() method to decrement
        // the waitgroup counter (count: 0)
        wg.Done()
    }(&wg)

    fmt.Println("waiting for waitgroup to unblock")

    // wait for the waitgroup to unblock (count: 1)
    wg.Wait()
    // (count: 0)
```

(*continued*)

```go
    fmt.Println("waitgroup is unblocked")

    if !completed {
        t.Fatal("waitgroup is not completed")
    }
}
```

```
$ go test -v -run Positive

=== RUN   Test_WaitGroup_Add_Positive
=== PAUSE Test_WaitGroup_Add_Positive
=== CONT  Test_WaitGroup_Add_Positive
waiting for waitgroup to unblock
done with waitgroup
waitgroup is unblocked
--- PASS: Test_WaitGroup_Add_Positive (0.01s)
PASS
ok      demo    0.351s
```

```
Go Version: go1.19
```

Adding a Zero Number

It is legal to call the `sync.WaitGroup.Add` method with a zero number, `0`, Listing 13.7. In this case, the `sync.WaitGroup.Add` method does nothing. The call becomes a no-op.

Listing 13.7 Adding a Zero Number of Goroutines

```go
func Test_WaitGroup_Add_Zero(t *testing.T) {
    t.Parallel()

    // create a new waitgroup (count: 0)
    var wg sync.WaitGroup

    // add 0 to the waitgroup (count: 0)
    wg.Add(0)
    // (count: 0)

    fmt.Println("waiting for waitgroup to unblock")

    // wait for the waitgroup to unblock (count: 0)
    // will not block since the counter is already 0
```

(continued)

```
    wg.Wait()
    // (count: 0)

    fmt.Println("waitgroup is unblocked")
}
```

```
$ go test -v -run Zero

=== RUN   Test_WaitGroup_Add_Zero
=== PAUSE Test_WaitGroup_Add_Zero
=== CONT  Test_WaitGroup_Add_Zero
waiting for waitgroup to unblock
waitgroup is unblocked
--- PASS: Test_WaitGroup_Add_Zero (0.00s)
PASS
ok      demo    0.166s
```

```
Go Version: go1.19
```

As you can see from the test output in Listing 13.7, the `sync.WaitGroup.Wait` method unblocked immediately because its internal counter is already zero.

Adding a Negative Number

When calling the `sync.WaitGroup.Add` method with a negative number, the `sync.WaitGroup.Add` method panics.

As you can see from the test output in Listing 13.8, the `sync.WaitGroup.Wait` method was never reached because the `sync.WaitGroup.Add` method panicked when we tried to add a negative number of goroutines.

Listing 13.8 Adding a Negative Number of Goroutines

```
func Test_WaitGroup_Add_Negative(t *testing.T) {
    t.Parallel()

    // create a new waitgroup (count: 0)
    var wg sync.WaitGroup

    // use an anonymous function to trap the panic
    // so we can properly mark the test as a failure
    func() {

        // defer a function to catch the panic
        defer func() {
```

(continued)

```
            // recover the panic
            if r := recover(); r != nil {
                // mark the test as a failure
                t.Fatal(r)
            }
        }()

        // add a negative number to the waitgroup
        // this will panic since the counter cannot be negative
        wg.Add(-1)

        fmt.Println("waiting for waitgroup to unblock")

        // this will never be reached
        wg.Wait()

        fmt.Println("waitgroup is unblocked")
    }()

}
```

```
$ go test -v -run Negative

=== RUN   Test_WaitGroup_Add_Negative
=== PAUSE Test_WaitGroup_Add_Negative
=== CONT  Test_WaitGroup_Add_Negative
    add_test.go:92: sync: negative WaitGroup counter
--- FAIL: Test_WaitGroup_Add_Negative (0.00s)
FAIL
exit status 1
FAIL    demo    0.753s
```

```
Go Version: go1.19
```

The Done Method

Once we increase that counter by calling the `sync.WaitGroup.Add` method, the `sync.WaitGroup.Wait` method blocks until we decrement the counter as we finish with each goroutine.

For each item we add to the `sync.WaitGroup` with the `sync.WaitGroup.Add` method, we need to call the `sync.WaitGroup.Done` method, Listing 13.9, to indicate that the goroutine is finished.

Listing 13.9 The `sync.WaitGroup.Done` method

```
$ go doc sync.WaitGroup.Done

package sync // import "sync"

func (wg *WaitGroup) Done()
    Done decrements the WaitGroup counter by one.
---------------------------------------------------------------------------
Go Version: go1.19
```

Consider Listing 13.10, which creates `N` goroutines and adds `N` to the `sync.WaitGroup` using the `sync.WaitGroup.Add` method. Each goroutine calls the `sync.WaitGroup.Done` method after it finishes. We then use the `sync.WaitGroup.Wait` method to wait for all of the goroutines to finish.

Listing 13.10 Testing the `sync.WaitGroup.Done` Method

```go
func Test_WaitGroup_Done(t *testing.T) {
    t.Parallel()

    const N = 5

    // create a new waitgroup (count: 0)
    var wg sync.WaitGroup

    // add 5 to the waitgroup (count: 5)
    wg.Add(N)

    for i := 0; i < N; i++ {

        // launch a goroutine that will call the
        // waitgroup's Done method when it finishes
        go func(i int) {

            // sleep briefly
            time.Sleep(time.Millisecond * time.Duration(i))

            fmt.Println("decrementing waiting by 1")

            // call the waitgroup's Done method
            // (count: count - 1)
            wg.Done()
```

(continued)

```
        }(i + 1)
    }

    fmt.Println("waiting for waitgroup to unblock")

    wg.Wait()

    fmt.Println("waitgroup is unblocked")
}
```

```
$ go test -v -timeout 1s

=== RUN   Test_WaitGroup_Done
=== PAUSE Test_WaitGroup_Done
=== CONT  Test_WaitGroup_Done
waiting for waitgroup to unblock
decremeting waiting by 1
decremeting waiting by 1
decremeting waiting by 1
decremeting waiting by 1
decremeting waiting by 1
waitgroup is unblocked
--- PASS: Test_WaitGroup_Done (0.01s)
PASS
ok      demo    0.384s
```

```
Go Version: go1.19
```

As we can see from the test output, Listing 13.10, the `sync.WaitGroup.Wait` method unblocked after all of the goroutines finished.

Improper Usage

> You *must* call `sync.WaitGroup.Done` exactly once for each number of items you add with `sync.WaitGroup.Add`.

If you don't call `sync.WaitGroup.Done` exactly once for each item you add with `sync.WaitGroup.Add`, the `sync.WaitGroup.Wait` method will block forever, which causes a deadlock and crashes your program, as shown in Listing 13.11.

Listing 13.11 Decrementing a `sync.WaitGroup` with the `sync.WaitGroup.Done` Method

```go
func Test_WaitGroup_Done(t *testing.T) {
    t.Parallel()

    const N = 5

    // create a new waitgroup (count: 0)
    var wg sync.WaitGroup

    // add 5 to the waitgroup (count: 5)
    wg.Add(N)

    for i := 0; i < N; i++ {

        // launch a goroutine that will call the
        // waitgroup's Done method when it finishes
        go func(i int) {

            // sleep briefly
            time.Sleep(time.Millisecond * time.Duration(i))

            fmt.Println("finished")

            // exiting with calling the Done method
            // (count: count)
        }(i + 1)
    }

    fmt.Println("waiting for waitgroup to unblock")

    // this will never unblock
    // because the goroutines never call Done
    // and the application will deadlock and panic
    wg.Wait()

    fmt.Println("waitgroup is unblocked")

}
```

```
$ go test -v -timeout 1s

=== RUN   Test_WaitGroup_Done
=== PAUSE Test_WaitGroup_Done
=== CONT  Test_WaitGroup_Done
```

(continued)

```
waiting for waitgroup to unblock
finished
finished
finished
finished
finished
panic: test timed out after 1s

goroutine 19 [running]:
testing.(*M).startAlarm.func1()
    /usr/local/go/src/testing/testing.go:2029 +0x8c
created by time.goFunc
    /usr/local/go/src/time/sleep.go:176 +0x3c

goroutine 1 [chan receive]:
testing.tRunner.func1()
    /usr/local/go/src/testing/testing.go:1405 +0x45c
testing.tRunner(0x140001361a0, 0x1400010fcb8)
    /usr/local/go/src/testing/testing.go:1445 +0x14c
testing.runTests(0x1400001e280?, {0x101045ea0, 0x1, 0x1},
➥{0x6e00000000000000?, 0x100e71218?, 0x10104e640?})
    /usr/local/go/src/testing/testing.go:1837 +0x3f0
testing.(*M).Run(0x1400001e280)
    /usr/local/go/src/testing/testing.go:1719 +0x500
main.main()
    _testmain.go:47 +0x1d0

goroutine 4 [semacquire]:
sync.runtime_Semacquire(0x0?)
    /usr/local/go/src/runtime/sema.go:56 +0x2c
sync.(*WaitGroup).Wait(0x14000012140)
    /usr/local/go/src/sync/waitgroup.go:136 +0x88
demo.Test_WaitGroup_Done(0x0?)
    ./done_test.go:43 0xd0
testing.tRunner(0x14000136340, 0x100fa1580)
    /usr/local/go/src/testing/testing.go:1439 +0x110
created by testing.(*T).Run
    /usr/local/go/src/testing/testing.go:1486 +0x300
exit status 2
FAIL    demo    1.225s
```

```
Go Version: go1.19
```

If you call `sync.WaitGroup.Done` more than the number of items you added with `sync.WaitGroup.Add`, the `sync.WaitGroup.Done` method panics, Listing 13.12. The result is the same as if you called `sync.WaitGroup.Add` with a negative number.

Listing 13.12 Panicking from Decrementing `sync.WaitGroup` Too Many Times

```
func Test_WaitGroup_Done(t *testing.T) {
    t.Parallel()

    func() {
        // defer a function to catch the panic
        defer func() {

            // recover the panic
            if r := recover(); r != nil {
                // mark the test as a failure
                t.Fatal(r)
            }
        }()

        // create a new waitgroup (count: 0)
        var wg sync.WaitGroup

        // call done creating a negative
        // waitgroup counter
        wg.Done()

        // this line is never reached
        fmt.Println("waitgroup is unblocked")
    }()

}
```

```
$ go test -v -timeout 1s

=== RUN   Test_WaitGroup_Done
=== PAUSE Test_WaitGroup_Done
=== CONT  Test_WaitGroup_Done
    done_test.go:20: sync: negative WaitGroup counter
--- FAIL: Test_WaitGroup_Done (0.00s)
FAIL
exit status 1
FAIL    demo    0.416s
```

```
Go Version: go1.19
```

Wrapping Up Wait Groups

Using a `sync.WaitGroup` is a great way to manage the number of goroutines or any other number of tests that need to finish before your program can continue.

As you can see, we can effectively use a `sync.WaitGroup` to manage the thumbnail generator goroutines from our initial example.

In Listing 13.13, we create a new `sync.WaitGroup`. Then, in the `for` loop, we use the `sync.WaitGroup.Add` method to add `1` to the `sync.WaitGroup`. We then pass a pointer to the `generateThumbnail` function to `sync.WaitGroup`. A pointer is needed because the `generateThumbnail` function needs to be able to modify the `sync.WaitGroup` by calling the `sync.WaitGroup.Done` method.

Finally, we call the `sync.WaitGroup.Wait` method to wait for all of the goroutines to finish.

Listing 13.13 Using a `sync.WaitGroup` to Manage the Thumbnail Generator Goroutines

```
func Test_ThumbnailGenerator(t *testing.T) {
    t.Parallel()

    // image that we need thumbnails for
    const image = "foo.png"

    var wg sync.WaitGroup

    // start 5 goroutines to generate thumbnails
    for i := 0; i < 5; i++ {
        wg.Add(1)

        // start a new goroutine for each thumbnail
        go generateThumbnail(&wg, image, i+1)

    }

    fmt.Println("Waiting for thumbnails to be generated")

    // wait for all goroutines to finish
    wg.Wait()

    fmt.Println("Finished generate all thumbnails")
}
```

The `generateThumbnail` function now receives a pointer to the `sync.WaitGroup` and defers a call to the `sync.WaitGroup.Done` method to indicate that the goroutine is finished when the function exits.

Finally, as you can see from our test output in Listing 13.14, the application now finishes successfully.

Listing 13.14 Generating Thumbnails Using a `sync.WaitGroup`

```
func generateThumbnail(wg *sync.WaitGroup, image string, size int) {
    defer wg.Done()

    // thumbnail to be generated
    thumb := fmt.Sprintf("%s@%dx.png", image, size)

    fmt.Println("Generating thumbnail:", thumb)

    // wait for the thumbnail to be ready
    time.Sleep(time.Millisecond * time.Duration(size))

    fmt.Println("Finished generating thumbnail:", thumb)
}
```

```
$ go test -v

=== RUN   Test_ThumbnailGenerator
=== PAUSE Test_ThumbnailGenerator
=== CONT  Test_ThumbnailGenerator
Waiting for thumbnails to be generated
Generating thumbnail: foo.png@5x.png
Generating thumbnail: foo.png@3x.png
Generating thumbnail: foo.png@4x.png
Generating thumbnail: foo.png@2x.png
Generating thumbnail: foo.png@1x.png
Finished generating thumbnail: foo.png@1x.png
Finished generating thumbnail: foo.png@2x.png
Finished generating thumbnail: foo.png@3x.png
Finished generating thumbnail: foo.png@4x.png
Finished generating thumbnail: foo.png@5x.png
Finished generate all thumbnails
--- PASS: Test_ThumbnailGenerator (0.01s)
PASS
ok      demo    0.310s
```

```
Go Version: go1.19
```

Error Management with Error Groups

One of the downsides of the `sync.WaitGroup` is that it has no error management built in to capture errors that occur in the goroutines. It also has an API that requires exact implementation; otherwise, it panics.

To help address some of these issues, the `golang.org/x/sync/errgroup`,[10] Listing 13.15, package was introduced, providing a simpler API as well as built-in error management.

Listing 13.15 The `golang.org/x/sync/errgroup` Package

```
$ go doc golang.org/x/sync/errgroup

package errgroup // import "golang.org/x/sync/errgroup"

Package errgroup provides synchronization, error propagation, and Context
➥cancelation for groups of goroutines working on subtasks of a common task.

type Group struct{ ... }
    func WithContext(ctx context.Context) (*Group, context.Context)
```

```
Go Version: go1.19
```

The Problem

Consider Listing 13.16. In the example, a number of goroutines are launched to call the `generateThumbnail` functions. A `sync.WaitGroup` is used to wait for all the goroutines to finish.

Listing 13.16 Managing Goroutines with `sync.WaitGroup`

```go
func Test_ThumbnailGenerator(t *testing.T) {
    t.Parallel()

    // image that we need thumbnails for
    const image = "foo.png"

    var wg sync.WaitGroup

    // start 5 goroutines to generate thumbnails
    for i := 0; i < 5; i++ {
        wg.Add(1)

        // start a new goroutine for each thumbnail
        go generateThumbnail(&wg, image, i+1)

    }
```

10. https://pkg.go.dev/golang.org/x/sync/errgroup

```
    fmt.Println("Waiting for thumbnails to be generated")

    // wait for all goroutines to finish
    wg.Wait()

    fmt.Println("Finished generate all thumbnails")
}
```

Inside the `generateThumbnail` function, Listing 13.17, we see that there is an error that occurs if the `size` argument is divisible by 5, which causes a panic, and the application crashes.

Listing 13.17 The `generateThumbnail` Function

```go
func generateThumbnail(wg *sync.WaitGroup, image string, size int) {
    defer wg.Done()

    // error if the size is divisible by 5
    if size%5 == 0 {
        // how do we return this error back to the main
        // goroutine without panicking?
        err := fmt.Errorf("%d is divisible by 5", size)
        panic(err)
    }

    // thumbnail to be generated
    thumb := fmt.Sprintf("%s@%dx.png", image, size)

    fmt.Println("Generating thumbnail:", thumb)

    // wait for the thumbnail to be ready
    time.Sleep(time.Millisecond * time.Duration(size))

    fmt.Println("Finished generating thumbnail:", thumb)
}
```

```
$ go test -v

=== RUN   Test_ThumbnailGenerator
=== PAUSE Test_ThumbnailGenerator
=== CONT  Test_ThumbnailGenerator
Waiting for thumbnails to be generated
panic: 5 is divisible by 5
```

(continued)

```
goroutine 9 [running]:
demo.generateThumbnail(0x0?, {0x10268ac77?, 0x0?}, 0x0?)
        ./demo_test.go:47 +0x21c
created by demo.Test_ThumbnailGenerator
        ./demo_test.go:24 +0x4c
exit status 2
FAIL    demo    0.619s
```

```
Go Version: go1.19
```

The Error Group

The `errgroup.Group`[11] type provides a minimal API when compared to the `sync.WaitGroup` type. There are only two methods: `errgroup.Group.Go`[12] and `errgroup.Group.Wait`.[13]

The `errgroup.Group` type, Listing 13.18, manages the counter for you, so there is no need for counter management as there is with the `sync.WaitGroup` type.

Listing 13.18 The `errgroup.Group` Type

```
$ go doc golang.org/x/sync/errgroup.Group

package errgroup // import "golang.org/x/sync/errgroup"

type Group struct {
        // Has unexported fields.
}
    A Group is a collection of goroutines working on subtasks that are part
    ➥of the same overall task.

    A zero Group is valid and does not cancel on error.

func WithContext(ctx context.Context) (*Group, context.Context)
func (g *Group) Go(f func() error)
func (g *Group) Wait() error
```

```
Go Version: go1.19
```

11. https://pkg.go.dev/golang.org/x/sync/errgroup#Group
12. https://pkg.go.dev/golang.org/x/sync/errgroup#Group.Go
13. https://pkg.go.dev/golang.org/x/sync/errgroup#Group.Wait

The `Go` Method

You use the `errgroup.Group.Go` method, Listing 13.19, to launch a goroutine for the provided `func() error` function provided. The `func() error` function is executed in a goroutine. If the function returns an error, the `errgroup.Group` type captures the error, cancels the other goroutines, and returns the error to the caller from the `errgroup.Group.Wait` method.

Listing 13.19 The `errgroup.Group.Go` Method

```
$ go doc golang.org/x/sync/errgroup.Group.Go

package errgroup // import "golang.org/x/sync/errgroup"

func (g *Group) Go(f func() error)
    Go calls the given function in a new goroutine.

    The first call to return a non-nil error cancels the group; its error
    ➥will be returned by Wait.

---------------------------------------------------------------------------
Go Version: go1.19
```

The `Wait` Method

You use `errgroup.Group.Wait` method, Listing 13.20, to wait for all the goroutines to finish. If any of the goroutines return an error, the `errgroup.Group` type returns the error to the caller.

Listing 13.20 The `errgroup.Group.Wait` Method

```
$ go doc golang.org/x/sync/errgroup.Group.Wait

package errgroup // import "golang.org/x/sync/errgroup"

func (g *Group) Wait() error
    Wait blocks until all function calls from the Go method have returned,
    ➥then returns the first non-nil error (if any) from them.

---------------------------------------------------------------------------
Go Version: go1.19
```

It is important to understand the `errgroup.Group.Wait` method returns *only* the *first* error that occurs. Any other errors that occur are ignored. In Listing 13.21, we are calling the `Go` method on an `errorgroup.Group` 10 times. Each time we pass a function that sleeps for a random time, prints a messages, and then returns an error. From the output of the test, you can see that the error returned by the `Wait` method was returned from function number 4. The other nine errors are discarded.

Listing 13.21 The `errgroup.Group.Wait` Method Returns Only the First Error

```go
func Test_ErrorGroup_Multiple_Errors(t *testing.T) {
    t.Parallel()

    var wg errgroup.Group

    for i := 0; i < 10; i++ {
        i := i + 1

        wg.Go(func() error {

            time.Sleep(time.Millisecond * time.Duration(rand.Intn(10)))

            fmt.Printf("about to error from %d\n", i)

            return fmt.Errorf("error %d", i)

        })
    }

    err := wg.Wait()
    if err != nil {
        t.Fatal(err)
    }

}
```

```
$ go test -v

=== RUN   Test_ErrorGroup_Multiple_Errors
=== PAUSE Test_ErrorGroup_Multiple_Errors
=== CONT  Test_ErrorGroup_Multiple_Errors
about to error from 4
about to error from 6
about to error from 3
about to error from 1
about to error from 9
about to error from 10
about to error from 8
about to error from 5
about to error from 2
about to error from 7
    demo_test.go:38: error 4
```

```
--- FAIL: Test_ErrorGroup_Multiple_Errors (0.01s)
FAIL
exit status 1
FAIL    demo    0.263s
```

```
Go Version: go1.19
```

Listening for Error Group Cancellation

When launching a number of goroutines, it can often be useful to let others know the tasks have all completed. The `errgroup.WithContext`[14] function, Listing 13.22, returns a new `errgroup.Group`, as well as a `context.Context` that can be listened to for cancellation.

Listing 13.22 The `errgroup.WithContext` Function

```go
func Test_ErrorGroup_Context(t *testing.T) {
    t.Parallel()

    // create a new error group
    // and a context that will be canceled
    // when the group is done
    wg, ctx := errgroup.WithContext(context.Background())

    // create a quit channel for the goroutine
    // waiting for the context to be canceled
    // can close to signal the goroutine has finished
    quit := make(chan struct{})

    // launch a goroutine that will
    // wait for the errgroup context to finish
    go func() {
        fmt.Println("waiting for context to cancel")

        // wait for the context to be canceled
        <-ctx.Done()

        fmt.Println("context canceled")

        // close the quit channel so the test
        // will finish
        close(quit)
```

(continued)

14. https://pkg.go.dev/golang.org/x/sync/errgroup#WithContext

```
	}()

	// add a task to the errgroup
	wg.Go(func() error {
		time.Sleep(time.Millisecond * 5)
		return nil
	})

	// wait for the errgroup to finish
	err := wg.Wait()
	if err != nil {
		t.Fatal(err)
	}

	// wait for the context goroutine to finish
	<-quit
}
```

```
$ go test -v

=== RUN   Test_ErrorGroup_Context
=== PAUSE Test_ErrorGroup_Context
=== CONT  Test_ErrorGroup_Context
waiting for context to cancel
context canceled
--- PASS: Test_ErrorGroup_Context (0.01s)
PASS
ok      demo    0.725s

Go Version: go1.19
```

Wrapping Up Error Groups

The `errgroup.Group` type provides a simpler API than the `sync.WaitGroup` type. It also provides built-in error management. This makes managing goroutines with error handling much easier. The downside is that it is not as flexible as the `sync.WaitGroup` type, where you are in charge of managing the counter.

Which type you choose to use will vary from situation to situation, so it is important to understand the tradeoffs and benefits of each before deciding which type to use.

Using a `errgroup.Group` allows you to clean up your code significantly to make it easier to understand and to manage errors.

As you can see from Listing 13.23, the `generateThumbnail` function no longer needs to take a `sync.WaitGroup` as an argument.

Listing 13.23 Update the `generateThumbnail` Function to use `errgroup.Group`

```
func generateThumbnail(image string, size int) error {

    // error if the size is divisible by 5
    if size%5 == 0 {
        return fmt.Errorf("%d is divisible by 5", size)
    }

    // thumbnail to be generated
    thumb := fmt.Sprintf("%s@%dx.png", image, size)

    fmt.Println("Generating thumbnail:", thumb)

    // wait for the thumbnail to be ready
    time.Sleep(time.Millisecond * time.Duration(size))

    fmt.Println("Finished generating thumbnail:", thumb)

    return nil
}
```

Being able to return an error from the function means the function no longer needs to panic.

As you can see from the output, Listing 13.24, the `generateThumbnail` function no longer panics, and the test is now able to exit properly.

Listing 13.24 Using the `errgroup.Group` Type

```
func Test_ThumbnailGenerator(t *testing.T) {
    t.Parallel()

    // image that we need thumbnails for
    const image = "foo.png"

    // create a new error group
    var wg errgroup.Group

    // start 5 goroutines to generate thumbnails
    for i := 0; i < 5; i++ {

        // capture the i to the current scope
        i := i
```

(continued)

```
        // start a new goroutine for each thumbnail
        wg.Go(func() error {

            // return the result of generateThumbnail
            return generateThumbnail(image, i)
        })

    }

    fmt.Println("Waiting for thumbnails to be generated")

    // wait for all goroutines to finish
    err := wg.Wait()

    // check for any errors
    if err != nil {
        t.Fatal(err)
    }

    fmt.Println("Finished generate all thumbnails")
}
```

```
$ go test -v

=== RUN   Test_ThumbnailGenerator
=== PAUSE Test_ThumbnailGenerator
=== CONT  Test_ThumbnailGenerator
Waiting for thumbnails to be generated
Generating thumbnail: foo.png@4x.png
Generating thumbnail: foo.png@3x.png
Generating thumbnail: foo.png@1x.png
Generating thumbnail: foo.png@2x.png
Finished generating thumbnail: foo.png@1x.png
Finished generating thumbnail: foo.png@2x.png
Finished generating thumbnail: foo.png@3x.png
Finished generating thumbnail: foo.png@4x.png
    demo_test.go:43: 0 is divisible by 5
--- FAIL: Test_ThumbnailGenerator (0.00s)
FAIL
exit status 1
FAIL    demo    0.570s
```

```
Go Version: go1.19
```

Data Races

When writing concurrent applications, it is common to run into what is called a race condition.[15] A race condition occurs when two different goroutines try to access the same shared resource.

Consider Listing 13.25. There are two different goroutines. One goroutine inserts values into a map, and the other goroutine ranges over the map and prints the values.

Listing 13.25 Two Goroutines Accessing a Shared Map

```
// launch a goroutine to
// write data in the map
go func() {
    for i := 0; i < 10; i++ {

        // loop putting data in the map
        data[i] = true
    }

    // cancel the context
    cancel()
}()

// launch a goroutine to
// read data from the map
go func() {
    // loop through the map
    // and print the keys/values
    for k, v := range data {
        fmt.Printf("%d: %v\n", k, v)
    }
}()
```

In Listing 13.26, we use those two goroutines to write a test to assert the map is written to and read from correctly.

Listing 13.26 A Passing Testing Without the Race Detector

```
func Test_Mutex(t *testing.T) {
    t.Parallel()
```

(continued)

15. https://en.wikipedia.org/wiki/Race_condition

```go
    // create a new cancellable context
    // to stop the test when the goroutines
    // are finished
    ctx := context.Background()
    ctx, cancel := context.WithTimeout(ctx, 20*time.Millisecond)
    defer cancel()

    // create a map to be used
    // as a shared resource
    data := map[int]bool{}

    // launch a goroutine to
    // write data in the map
    go func() {
        for i := 0; i < 10; i++ {

            // loop putting data in the map
            data[i] = true
        }

        // cancel the context
        cancel()
    }()

    // launch a goroutine to
    // read data from the map
    go func() {
        // loop through the map
        // and print the keys/values
        for k, v := range data {
            fmt.Printf("%d: %v\n", k, v)
        }
    }()

    // wait for the context to be canceled
    <-ctx.Done()

    if len(data) != 10 {
        t.Fatalf("expected 10 items in the map, got %d", len(data))
    }
}
```

```
$ go test -v

=== RUN   Test_Mutex
=== PAUSE Test_Mutex
=== CONT  Test_Mutex
--- PASS: Test_Mutex (0.00s)
PASS
ok      demo    0.471s
```

```
Go Version: go1.19
```

A quick glance at the test output, Listing 13.26, would seem to imply that the tests have passed successfully, but this is not the case.

The Race Detector

A few of the Go commands, such as `test` and `build`, have the `-race` flag exposed. When used, the `-race` flag tells the Go compiler to create a special version of the binary or test binary that will detect and report race conditions.

If we run the test again, this time with the `-race` flag, we get a *very* different result, as shown in Listing 13.27.

Listing 13.27 Tests Failing with the Race Detector

```
$ go test -v -race

=== RUN   Test_Mutex
=== PAUSE Test_Mutex
=== CONT  Test_Mutex
--- PASS: Test_Mutex (0.00s)
==================
WARNING: DATA RACE
Read at 0x00c00011c3f0 by goroutine 9:
  runtime.mapdelete()
      /usr/local/go/src/runtime/map.go:695 +0x46c
  demo.Test_Mutex.func2()
      ./demo_test.go:46 +0x50

Previous write at 0x00c00011c3f0 by goroutine 8:
  runtime.mapaccess2_fast64()
      /usr/local/go/src/runtime/map_fast64.go:53 +0x1cc
  demo.Test_Mutex.func1()
      ./demo_test.go:32 +0x50
```

(*continued*)

```
Goroutine 9 (running) created at:
  demo.Test_Mutex()
      ./demo_test.go:43 +0x188
  testing.tRunner()
      /usr/local/go/src/testing/testing.go:1439 +0x18c
  testing.(*T).Run.func1()
      /usr/local/go/src/testing/testing.go:1486 +0x44

Goroutine 8 (finished) created at:
  demo.Test_Mutex()
      ./demo_test.go:28 +0x124
  testing.tRunner()
      /usr/local/go/src/testing/testing.go:1439 +0x18c
  testing.(*T).Run.func1()
      /usr/local/go/src/testing/testing.go:1486 +0x44
==================
FAIL
exit status 1
FAIL    demo    0.962s
```

```
Go Version: go1.19
```

As you can see from the output, the Go race detector found a race condition in our code.

If we examine the top two entries in a race condition warning, Listing 13.28, it tells us where the two conflicting lines of code are.

Listing 13.28 Reading the Race Detector Output

```
Read at 0x00c00018204b by goroutine 9:
  demo.Test_Mutex.func2()
      problem/demo_test.go:46 +0xa5

Previous write at 0x00c00018204b by goroutine 8:
  demo.Test_Mutex.func1()
      problem/demo_test.go:32 +0x5c
```

A read of the shared resource was happening at `demo_test.go:46`, and a write was happening at `demo_test.go:32`. We need to synchronize or lock these two goroutines so that they don't both try to access the shared resource at the same time.

Most, but Not All

The Go race detector makes a simple guarantee with you (the end user).

> The Go race detector may not find *all* the race conditions in your code, but the ones it does find are *real* and *must* be fixed.

A race condition *will* panic and crash your application. If the race detector finds a race condition, you *must* fix it.

Wrapping Up the Race Detector

The race detector is an *invaluable* tool when developing Go applications. When running tests with the `-race` flag, you will notice a slowdown in test performance. The race detector has to do a lot of work to track those conditions.

> *Always* enable the `-race` flag on your CI, such as GitHub Actions.

Once identified, the `sync` package, Listing 13.29, provides a number of ways that you can fix issues.

Listing 13.29 The `sync` Package

```
$ go doc -short sync

type Cond struct{ ... }
    func NewCond(l Locker) *Cond
type Locker interface{ ... }
type Map struct{ ... }
type Mutex struct{ ... }
type Once struct{ ... }
type Pool struct{ ... }
type RWMutex struct{ ... }
type WaitGroup struct{ ... }

---------------------------------------------------------

Go Version: go1.19
```

Synchronizing Access with a Mutex

When you run tests with the `-race` flag, Go's built-in race detector can help you find data races in your code. In Listing 13.30, for example, tests that pass normally fail when run with the race detector. The failure message lists the data race found and where the reads and writes occur in our code.

Listing 13.30 Detecting Race Conditions in Tests

```
$ go test -v -race

=== RUN   Test_Mutex
=== PAUSE Test_Mutex
```

(continued)

```
=== CONT  Test_Mutex
--- PASS: Test_Mutex (0.00s)
PASS
==================
WARNING: DATA RACE
Read at 0x00c00019e3f0 by goroutine 9:
  runtime.mapdelete()
      /usr/local/go/src/runtime/map.go:695 +0x46c
  demo.Test_Mutex.func2()
      ./demo_test.go:46 +0x50

Previous write at 0x00c00019e3f0 by goroutine 8:
  runtime.mapaccess2_fast64()
      /usr/local/go/src/runtime/map_fast64.go:53 +0x1cc
  demo.Test_Mutex.func1()
      ./demo_test.go:32 +0x50

Goroutine 9 (running) created at:
  demo.Test_Mutex()
      ./demo_test.go:43 +0x188
  testing.tRunner()
      /usr/local/go/src/testing/testing.go:1439 +0x18c
  testing.(*T).Run.func1()
      /usr/local/go/src/testing/testing.go:1486 +0x44

Goroutine 8 (finished) created at:
  demo.Test_Mutex()
      ./demo_test.go:28 +0x124
  testing.tRunner()
      /usr/local/go/src/testing/testing.go:1439 +0x18c
  testing.(*T).Run.func1()
      /usr/local/go/src/testing/testing.go:1486 +0x44
==================
0: true
2: true
4: true
7: true
9: true
1: true
3: true
5: true
6: true
8: true
```

```
Found 1 data race(s)
exit status 66
FAIL    demo    0.862s
```

```
Go Version: go1.19
```

In the test in Listing 13.31, we have two different goroutines. The first is modifying a shared resource—in this case, a map. The second goroutine is ranging over the map and printing out the map's values.

In order for us to be able to fix this race condition, we need to be able to synchronize access to the shared resource.

Listing 13.31 Two Goroutines Accessing a Shared Map

```go
// launch a goroutine to
// read data from the map
go func() {
    // loop through the map
    // and print the keys/values
    for k, v := range data {
        fmt.Printf("%d: %v\n", k, v)
    }
}()

// launch a goroutine to
// put data in the map
go func() {
    for i := 0; i < 10; i++ {

        // loop putting data in the map
        data[i] = true
    }

    // cancel the context
    cancel()
}()
```

Locker

To synchronize access to the shared resource, we need to be able to lock access to the resource. By locking a shared resource, we can ensure that only one goroutine at a time can access the resource and that the resource is not modified by another goroutine while it is locked.

The `sync.Locker`[16] interface, Listing 13.32, defines the methods that a type must implement to be able to lock and unlock a shared resource.

Listing 13.32 The `sync.Locker` Interface

```
$ go doc sync.Locker

package sync // import "sync"

type Locker interface {
        Lock()
        Unlock()
}
    A Locker represents an object that can be locked and unlocked.

---------------------------------------------------------------
Go Version: go1.19
```

Locker Methods

You can use the `sync.Locker.Lock`[17] method, Listing 13.33, to lock the shared resource. Once a resource is locked, no other goroutine can access the resource until it is unlocked.

Listing 13.33 The `sync.Locker.Lock` Method

```
$ go doc sync.Mutex.Lock

package sync // import "sync"

func (m *Mutex) Lock()
    Lock locks m. If the lock is already in use, the calling goroutine blocks
    ➥until the mutex is available.

---------------------------------------------------------------
Go Version: go1.19
```

You can use the `sync.Locker.Unlock`[18] method, Listing 13.34, to unlock the shared resource. Once a resource is unlocked, other goroutines can access the resource.

16. https://pkg.go.dev/sync#Locker
17. https://pkg.go.dev/sync#Locker.Lock
18. https://pkg.go.dev/sync#Locker.Unlock

Listing 13.34 The sync.Locker.Unlock Method

```
$ go doc sync.Mutex.Unlock

package sync // import "sync"

func (m *Mutex) Unlock()
    Unlock unlocks m. It is a run-time error if m is not locked on entry to
    ➥Unlock.

    A locked Mutex is not associated with a particular goroutine. It is
    ➥allowed for one goroutine to lock a Mutex and then arrange for another
    ➥goroutine to unlock it.
```

```
Go Version: go1.19
```

Using a Mutex

The most basic mutex available in Go is the sync.Mutex type, Listing 13.35. The sync.Mutex uses a basic binary semaphore lock. This means that only one goroutine can access the resource at a time.

Listing 13.35 The sync.Mutex Type

```
$ go doc sync.Mutex

package sync // import "sync"

type Mutex struct {
        // Has unexported fields.
}
    ➥A Mutex is a mutual exclusion lock. The zero value for a Mutex is an
    ➥unlocked mutex.

    A Mutex must not be copied after first use.

func (m *Mutex) Lock()
func (m *Mutex) TryLock() bool
func (m *Mutex) Unlock()
```

```
Go Version: go1.19
```

To use a sync.Mutex, you need to wrap the areas of code that you want to synchronize access to by first locking the sync.Mutex and then unlocking it. For example, in the second goroutine in Listing 13.36, a mutex is being used to lock access around writing values into the `data` map.

Listing 13.36 Locking Resources with a `sync.Mutex`

```go
// launch a goroutine to
// read data from the map
go func() {
    // lock the mutex
    mu.Lock()

    // loop through the map
    // and print the keys/values
    for k, v := range data {
        fmt.Printf("%d: %v\n", k, v)
    }

    // unlock the mutex
    mu.Unlock()
}()

// launch a goroutine to
// put data in the map
go func() {

    for i := 0; i < 10; i++ {
        // lock the mutex
        mu.Lock()

        // loop putting data in the map
        data[i] = true

        // unlock the mutex
        mu.Unlock()
    }

    // cancel the context
    cancel()
}()
```

By locking access to the shared resource, you can ensure that only one goroutine at a time can access the resource. Our test output, Listing 13.37, confirms that the shared resource is only accessed by one goroutine at a time with a successful exit.

Listing 13.37 Passing Race Detector Tests

```
$ go test -v -race

=== RUN   Test_Mutex
=== PAUSE Test_Mutex
=== CONT  Test_Mutex
9: true
--- PASS: Test_Mutex (0.00s)
0: true
2: true
5: true
8: true
7: true
1: true
PASS
3: true
4: true
6: true
ok      demo    0.810s

--------------------------------------------------------------------------------
Go Version: go1.19
```

RWMutex

Often, applications read a shared resource instead of writing to them. The `sync.Mutex` is a very heavy-weight locking mechanism. Access to a shared resource, whether it be a read or write, is blocked until the resource is unlocked. Only one goroutine can access a shared resource at a time.

When you want to be able to both read and write to a shared resource, you need to use a `sync.RWMutex`, Listing 13.38. The `sync.RWMutex` is a lighter-weight locking mechanism. A `sync.RWMutex` can allow many goroutines to read from the resource at the same time, but only one goroutine can write to the resource at a time.

Listing 13.38 The `sync.RWMutex` Type

```
$ go doc sync.RWMutex

package sync // import "sync"

type RWMutex struct {
        // Has unexported fields.
}
```

(continued)

```
    A RWMutex is a reader/writer mutual exclusion lock. The lock can be held
    ➥by an arbitrary number of readers or a single writer. The zero value for
    ➥a RWMutex is an unlocked mutex.

    A RWMutex must not be copied after first use.

    If a goroutine holds a RWMutex for reading and another goroutine might
    ➥call Lock, no goroutine should expect to be able to acquire a read lock
    ➥until the initial read lock is released. In particular, this prohibits
    ➥recursive read locking. This is to ensure that the lock eventually
    ➥becomes available; a blocked Lock call excludes new readers from
    ➥acquiring the lock.

func (rw *RWMutex) Lock()
func (rw *RWMutex) RLock()
func (rw *RWMutex) RLocker() Locker
func (rw *RWMutex) RUnlock()
func (rw *RWMutex) TryLock() bool
func (rw *RWMutex) TryRLock() bool
func (rw *RWMutex) Unlock()

--------------------------------------------------------------------------------
Go Version: go1.19
```

The `sync.RWMutex` offers two additional methods beyond those of the `sync.Locker` interface. You can use the `sync.RWMutex.Rlock`[19] and `sync.RWMutex.RUnlock`[20] methods to lock the resource for reading. The `sync.RWMutex.Lock`[21] and `sync.RWMutex.Unlock`[22] methods are used to lock the resource across all goroutines for writing.

In Listing 13.39, we update the goroutine that is reading the resource to use the `sync.RWMutex.Rlock` method instead of the `sync.Mutex.Lock` method. This will allow for many goroutines to read from the resource at the same time.

Listing 13.39 Using a `sync.RWMutex`

```go
// launch a goroutine to
// read data from the map
go func() {
    // lock the mutex
    mu.RLock()
```

19. https://pkg.go.dev/sync#RWMutex.Rlock
20. https://pkg.go.dev/sync#RWMutex.RUnlock
21. https://pkg.go.dev/sync#RWMutex.Lock
22. https://pkg.go.dev/sync#RWMutex.Unlock

```
    // loop through the map
    // and print the keys/values
    for k, v := range data {
        fmt.Printf("%d: %v\n", k, v)
    }

    // unlock the mutex
    mu.RUnlock()
}()
```

```
$ go test -v -race

=== RUN   Test_RWMutex
=== PAUSE Test_RWMutex
=== CONT  Test_RWMutex
4: true
5: true
6: true
--- PASS: Test_RWMutex (0.00s)
7: true
9: true
0: true
1: true
PASS
8: true
2: true
3: true
ok      demo    0.917s
```

```
Go Version: go1.19
```

The tests in Listing 13.39 continue to pass, but the performance of the program is improved by allowing multiple goroutines to read from the shared resource instead of arbitrarily locking all goroutines all at once.

Improper Usage

When using either `sync.Mutex` or `sync.RWMutex`, you must take care in making sure to lock and unlock in the proper order.

Consider Listing 13.40. We have a `sync.Mutex`, and we attempt to call `sync.Mutex.Lock` twice.

Listing 13.40 Attempting to Lock a `sync.Mutex` Twice

```
func Test_Mutex_Locks(t *testing.T) {
    t.Parallel()

    // create a new mutex
    var mu sync.Mutex

    // lock the mutex
    mu.Lock()

    fmt.Println("locked. locking again.")

    // try to lock the mutex again
    // this will block/deadlock
    // because the mutex is already locked
    // and the lock was not released
    mu.Lock()

    fmt.Println("unlocked twice")
}
```

The result is the program will deadlock and crash, as shown in Listing 13.41. The reason is that a call to `sync.Mutex.Lock` blocks until the `sync.Mutex.Unlock` method is called. Because we have already locked the `sync.Mutex`, the second call to `sync.Mutex.Lock` blocks indefinitely because it is never unlocked.

Listing 13.41 A Panic while Trying to Unlock an Already-Unlocked `sync.Mutex`

```
$ go test -v -timeout 10ms

=== RUN   Test_Mutex_Locks
=== PAUSE Test_Mutex_Locks
=== CONT  Test_Mutex_Locks
locked. locking again.
panic: test timed out after 10ms

goroutine 33 [running]:
testing.(*M).startAlarm.func1()
        /usr/local/go/src/testing/testing.go:2029 +0x8c
created by time.goFunc
        /usr/local/go/src/time/sleep.go:176 +0x3c
```

```
goroutine 1 [chan receive]:
testing.tRunner.func1()
        /usr/local/go/src/testing/testing.go:1405 +0x45c
testing.tRunner(0x140001361a0, 0x1400010fcb8)
        /usr/local/go/src/testing/testing.go:1445 +0x14c
testing.runTests(0x1400001e1e0?, {0x100ec9ea0, 0x1, 0x1},
➥{0xe000000000000000?, 0x100cf5218?, 0x100ed2640?})
        /usr/local/go/src/testing/testing.go:1837 +0x3f0

testing.(*M).Run(0x1400001e1e0)
        /usr/local/go/src/testing/testing.go:1719 +0x500
main.main()
        _testmain.go:47 +0x1d0

goroutine 4 [semacquire]:
sync.runtime_SemacquireMutex(0x1400000e018?, 0x20?, 0x17?)
        /usr/local/go/src/runtime/sema.go:71 +0x28
sync.(*Mutex).lockSlow(0x14000012140)
        /usr/local/go/src/sync/mutex.go:162 +0x180
sync.(*Mutex).Lock(...)
        /usr/local/go/src/sync/mutex.go:81
demo.Test_Mutex_Locks(0x0?)
        ./demo_test.go:25 +0x130
testing.tRunner(0x14000136340, 0x100e25298)
        /usr/local/go/src/testing/testing.go:1439 +0x110
created by testing.(*T).Run
        /usr/local/go/src/testing/testing.go:1486 +0x300
exit status 2
FAIL    demo    0.527s
```

```
Go Version: go1.19
```

Worse than a deadlock caused by waiting for a lock that will never be unlocked is unlocking a lock that has not been locked.

In Listing 13.42, the result is a fatal error that crashes the application. The reason is that we have not locked the `sync.Mutex` before attempting to unlock it.

Listing 13.42 A Panic while Trying to Unlock an Unlocked `sync.Mutex`

```go
func Test_Mutex_Unlock(t *testing.T) {
    t.Parallel()

    // create a new mutex
    var mu sync.Mutex
```

(continued)

```
    // unlock the mutex
    mu.Unlock()
}
```

```
$ go test -v

=== RUN   Test_Mutex_Unlock
=== PAUSE Test_Mutex_Unlock
=== CONT  Test_Mutex_Unlock
fatal error: sync: unlock of unlocked mutex

goroutine 18 [running]:
runtime.throw(0x104573b02?, 0x1400005af18?)
        /usr/local/go/src/runtime/panic.go:992 +0x50 fp=0x1400005aee0
        ➥sp=0x1400005aeb0 pc=0x1044ba9a0
sync.throw(0x104573b02?, 0x1045b6260?)
        /usr/local/go/src/runtime/panic.go:978 +0x24 fp=0x1400005af00
        ➥sp=0x1400005aee0 pc=0x1044e5664
sync.(*Mutex).unlockSlow(0x140001280b0, 0xffffffff)
        /usr/local/go/src/sync/mutex.go:220 +0x3c fp=0x1400005af30
        ➥sp=0x1400005af00 pc=0x1044ef44c
sync.(*Mutex).Unlock(...)
        /usr/local/go/src/sync/mutex.go:214
demo.Test_Mutex_Unlock(0x0?)
        ./demo_test.go:16 +0x74 fp=0x1400005af60 sp=0x1400005af30
        ➥pc=0x10456d794
testing.tRunner(0x1400010b380, 0x1045c9298)
        /usr/local/go/src/testing/testing.go:1439 +0x110 fp=0x1400005afb0
        ➥sp=0x1400005af60 pc=0x104537660
testing.(*T).Run.func1()
        /usr/local/go/src/testing/testing.go:1486 +0x30 fp=0x1400005afd0
        ➥sp=0x1400005afb0 pc=0x1045383d0
runtime.goexit()
        /usr/local/go/src/runtime/asm_arm64.s:1263 +0x4 fp=0x1400005afd0
        ➥sp=0x1400005afd0 pc=0x1044ea2a4
created by testing.(*T).Run
        /usr/local/go/src/testing/testing.go:1486 +0x300

goroutine 1 [chan receive]:
testing.tRunner.func1()
        /usr/local/go/src/testing/testing.go:1405 +0x45c
testing.tRunner(0x1400010b1e0, 0x14000131cb8)
        /usr/local/go/src/testing/testing.go:1445 +0x14c
testing.runTests(0x140001421e0?, {0x10466dea0, 0x1, 0x1}, {0xa500000000000000?,
```

```
➥0x104499218?, 0x104676640?})
        /usr/local/go/src/testing/testing.go:1837 +0x3f0
testing.(*M).Run(0x140001421e0)
        /usr/local/go/src/testing/testing.go:1719 +0x500
main.main()
        _testmain.go:47 +0x1d0

exit status 2
FAIL    demo    0.260s
```

```
Go Version: go1.19
```

Wrapping Up Read/Write Mutexes

While there are pitfalls and areas of concern when using mutexes, such as deadlocks and improper usage, `sync.Mutex` and `sync.RWMutex` are excellent tools for protecting shared resources. They are also the most commonly used locking mechanisms in Go.

Performing Tasks Only Once

There are many times when you want to perform a task only once. For example, you might want to create a database connection only once and then use it to perform a number of queries. You can use the `sync.Once`[23] type to do this.

As you can see from the documentation in Listing 13.43, the use of `sync.Once` is very simple. You just need to create a variable of type `sync.Once` and then call the `sync.Once.Do`[24] method with a function that you want to run only once.

Listing 13.43 The `sync.Once` Type

```
$ go doc -all sync.Once

package sync // import "sync"

type Once struct {
        // Has unexported fields.
}
    Once is an object that will perform exactly one action.

    A Once must not be copied after first use.
```

(*continued*)

23. https://pkg.go.dev/sync#Once
24. https://pkg.go.dev/sync#Once.Do

```
func (o *Once) Do(f func())
    Do calls the function f if and only if Do is being called for the first
    ➥time for this instance of Once. In other words, given

        var once Once

    if once.Do(f) is called multiple times, only the first call will invoke f,
    ➥even if f has a different value in each invocation. A new instance of
    ➥Once is required for each function to execute.

    Do is intended for initialization that must be run exactly once. Since f
    ➥is niladic, it may be necessary to use a function literal to capture the
    ➥arguments to a function to be invoked by Do:

        config.once.Do(func() { config.init(filename) })

    Because no call to Do returns until the one call to f returns, if f
    ➥causes Do to be called, it will deadlock.

    If f panics, Do considers it to have returned; future calls of Do return
    ➥without calling f.
---------------------------------------------------------------------------
Go Version: go1.19
```

The Problem

Often we want to use `sync.Once` to perform some heavy, expensive tasks only once.

Consider Listing 13.44. The `Build` function can be called many times, but we only want it to run once because it takes some time to complete.

Listing 13.44 The `Build` Method Is Slow and Should Be Called Only Once

```go
type Builder struct {
    Built bool
}

func (b *Builder) Build() error {

    fmt.Print("building...")

    time.Sleep(10 * time.Millisecond)

    fmt.Println("built")
```

```
    b.Built = true

    // validate the message
    if !b.Built {
        return fmt.Errorf("expected builder to be built")
    }

    // return the b.msg and the error variable
    return nil
}
```

As you can see from the test output, Listing 13.45, every call to the `Build` function takes a long time to complete, and each call performs the same task.

Listing 13.45 Output Confirming the `Build` Function Runs Every Time It Is Called

```
func Test_Once(t *testing.T) {
    t.Parallel()

    b := &Builder{}

    for i := 0; i < 5; i++ {

        err := b.Build()
        if err != nil {
            t.Fatal(err)
        }

        fmt.Println("builder built")

        if !b.Built {
            t.Fatal("expected builder to be built")
        }
    }
}
```

```
$ go test -v

=== RUN   Test_Once
=== PAUSE Test_Once
=== CONT  Test_Once
building...built
builder built
```

(continued)

```
building...built
builder built

building...built
builder built
building...built
builder built
building...built
builder built
--- PASS: Test_Once (0.05s)
PASS
ok      demo    0.265s
```

```
Go Version: go1.19
```

Implementing Once

As shown in Listing 13.46, you can use the `sync.Once` type inside the `Build` function to ensure that the expensive task is only performed once.

Listing 13.46 Using `sync.Once` to Run a Function Once

```go
type Builder struct {
    Built bool
    once  sync.Once
}

func (b *Builder) Build() error {

    var err error

    b.once.Do(func() {

        fmt.Print("building...")

        time.Sleep(10 * time.Millisecond)

        fmt.Println("built")

        b.Built = true

        // validate the message
        if !b.Built {
```

```
            err = fmt.Errorf("expected builder to be built")
        }
    })

    // return the b.msg and the error variable
    return err
}
```

As you can see from the test output, Listing 13.47, the `Build` function now performs the expensive task only once, and subsequent calls to the function are very fast.

Listing 13.47 Output Confirming the `Build` Function Runs Only Once

```
$ go test -v

=== RUN   Test_Once
=== PAUSE Test_Once
=== CONT  Test_Once
building...built
builder built
builder built
builder built
builder built
builder built
--- PASS: Test_Once (0.01s)
PASS
ok      demo    0.248s

Go Version: go1.19
```

Closing Channels with Once

The `sync.Once` type is useful for closing channels. When you want to close a channel, you need to ensure that the channel is closed only once. If you try to close the channel more than once, you get a panic, and the program crashes.

Consider the example in Listing 13.48. The `Quit` method on the `Manager` is in charge of closing the `quit` channel when the `Manager` is no longer needed.

Listing 13.48 If Called Repeatedly, the `Quit` Function Panics and Closes an Already-Closed Channel

```
type Manager struct {
    quit chan struct{}
}

func (m *Manager) Quit() {
    fmt.Println("closing quit channel")
    close(m.quit)
}
```

If, however, the `Quit` method is called more than once, we are trying to close the channel more than once. We get a panic, and the program crashes.

As you can see in Listing 13.49, the tests failed as a result of trying to close the channel more than once and caused a panic.

Listing 13.49 Panicking When Trying to Close a Channel Multiple Times

```
func Test_Closing_Channels(t *testing.T) {
    t.Parallel()

    func() {
        // defer a function to catch the panic
        defer func() {

            // recover the panic
            if r := recover(); r != nil {
                // mark the test as a failure
                t.Fatal(r)
            }
        }()

        m := &Manager{
            quit: make(chan struct{}),
        }

        // close the manager's quit channel
        m.Quit()

        // try to close the manager's quit channel again
        // this will panic
        m.Quit()
```

```
    }()
}
```

```
$ go test -v

=== RUN   Test_Closing_Channels
=== PAUSE Test_Closing_Channels
=== CONT  Test_Closing_Channels
closing quit channel
closing quit channel
    demo_test.go:31: close of closed channel
--- FAIL: Test_Closing_Channels (0.00s)
FAIL
exit status 1
FAIL    demo    0.667s
```

```
Go Version: go1.19
```

In Listing 13.50, we use the `sync.Once` type to ensure that the `Quit` method, regardless of how many times it is called, only closes the channel once.

Listing 13.50 Using `sync.Once` to Close a Channel Only Once

```go
type Manager struct {
    quit chan struct{}
    once sync.Once
}

func (m *Manager) Quit() {

    // close the manager's quit channel
    // this will only close the channel once
    m.once.Do(func() {
        fmt.Println("closing quit channel")
        close(m.quit)
    })
}
```

As you can see from the test output, Listing 13.51, the `Quit` method now closes the channel only once, and subsequent calls to the `Quit` method have no effect.

Listing 13.51 Output Confirming the `Quit` Method Closes the Channel Only Once

```
func Test_Closing_Channels(t *testing.T) {
    t.Parallel()

    m := &Manager{
        quit: make(chan struct{}),
    }

    // close the manager's quit channel
    m.Quit()

    // try to close the manager's quit channel again
    // this will now have no effect
    m.Quit()
}
```

```
$ go test -v

=== RUN   Test_Closing_Channels
=== PAUSE Test_Closing_Channels
=== CONT  Test_Closing_Channels

closing quit channel
--- PASS: Test_Closing_Channels (0.00s)
PASS
ok      demo    0.523s
```

```
Go Version: go1.19
```

Summary

In this chapter, we took a look at just a few of the synchronization types and functions in Go. First, we explored how to use a `sync.WaitGroup` to wait for a number of goroutines to finish. Then we explained how to use `sync.ErrGroup` to wait for a number of goroutines to finish and return an error if any of them failed. Next, we discussed how to use `sync.Mutex` and `sync.RWMutex` to synchronize access to a shared resource. Finally, we covered how to use `sync.Once` to ensure a function is only executed one time.

Index

SYMBOLS

A

B

C

D

E

F

G

H

I

J

K

L

M

T

U

Z