Programmer's guide to idiomatic and testable code

Go by Example

İnanç Gümüş

MEAP

MANNING

Programmer's guide to idiomatic and testable code

Go by Example

İnanç Gümüş

MEAP

MANNING

Go by Example

welcome

Thanks for purchasing the MEAP for *Go by Example: Programmer's guide to idiomatic and testable code*. If you want to write Go code that's easy to maintain and extend, you've come to the right place!

To get the most benefit from this book, you'll need to be an experienced developer with an intermediate knowledge of testing in any language. You may be new to Go, but knowing the basics will help along the way.

As I write this, there's no book yet on testing in Go, but there's a growing need for developers. Go is a young language, and hundreds of thousands of developers from other programming languages come to Go every year. If you're one of them, you may be wondering how to transfer your existing knowledge of other languages, particularly when it comes to testing.

This book will teach you how to implement practical, real-life Go projects with tests from scratch. I'll be showing you different testing approaches—what to do, what not to do, and when and why you should use a particular method using real-life, practical examples you might encounter in the workplace.

I got the idea for this book based on questions I received from folks who've taken my Udemy course or are following my blog. *Idiomatic Go* will answer these questions and more:

- How can I test in Go?
- How can I test my program?
- What should I test—or not test?
- Am I testing correctly? What are the best practices?
- What makes Go tests and tests in other programming languages different?
- How can I build an easy to maintain Go program from scratch with tests?

All said, I hope you enjoy this book and that you find it useful to your own programming practice. If you have any questions, comments, or suggestions, please share them in Manning’s liveBook Discussion forum for my book. And, if you want to stay up to date on all the latest Go test best practices, be sure to give my Twitter a follow.

Let’s get started with Go testing together!

—Inanc Gumus

In this book

1 Getting Started

This chapter covers

- Importance of writing idiomatic and testable code
- Introducing Go's big features

Go is a modern and simple programming language that makes it convenient for individuals and distributed teams to work together to develop efficient, adaptable, maintainable, and scalable software that harnesses the full power of today's multi-core CPU systems. It's a perfect language for developing command-line tools, network programs, and so on.

This chapter explores the book's goals, introduces Go, and showcases Go's big features without fully delving into Go's mechanics and idioms (I'll leave the details to other chapters).

Let's take a look at why you should read this book and what you can expect from it.

1.1 Why should you read this book?

Getting up to speed with Go is so straightforward that in 2020, I tweeted:

> "Go is easy to learn. Hard to master."

While most programming languages can be difficult to master, Go is one of the few that is relatively easy to learn. Experienced programmers can learn the basics of Go in just a week. Yet, using Go to its full potential requires a deep understanding of its idioms and mechanics.

So, this book has two goals:

1. Crafting idiomatic and readable in Go.
2. Crafting testable and maintainable code in Go.

Rather than the usual boring examples like foo, bar, and baz, or Person, Animal, etc. this book aims to teach how to craft idiomatic and testable Go code from scratch with real-world examples. With this approach of learning by examples, you will clearly see why and when to use Go's features without worrying about making common mistakes beginners make.

Together, we'll explore command-line tools, design concurrent programs, structure HTTP servers, and clients, and effectively use interfaces to write adaptable code.

Each chapter includes examples and exercises that allow you to code along with the book and cement what you learn. In my thirty years of programming experience, that's the best way to learn another programming language or anything else.

I'm writing this book for intermediate-level programmers experienced in another language but new to Go. Those with experience in Go might also find what the book explores enlightening. I won't talk much about the basics of programming, so you also need basic programming skills like arrays, hash tables, etc., but you don't need to be a computer scientist. Knowing a bit or two about computer CPU and memory work will help.

Let's continue with this book's primary goals: crafting idiomatic and testable code. Then, we'll dive into Go's distinctive and flagship features. By the end of this book, you'll be knowledgeable enough to start using Go effectively in your projects.

Note

You can follow me for Go tips, tricks, and updates: https://x.com/inancgumus

1.2 Crafting idiomatic code

> "Programs are meant to be read by humans and only incidentally for computers to execute."

— H. Abelson and G.J. Sussman, from the book Structure and Interpretation

of Computer Programs

Writing complex code is easy. However, achieving simplicity is not. The puzzle's missing piece is combining the correct pieces to craft meaningful, expressive, yet powerful code.

The good news is that there is a recommended approach called "*Idiomatic Go*," which means the best practices for crafting Go code. By following idiomatic practices, you can improve the maintainability and understandability of your code.

Here is what idiomatic Go code looks like:

- *Simple*—Code is straightforward to understand.
- *Readable*—Code is easy to read (readability is better than clever code).
- *Explicit*—Code can be read from top to bottom without making many assumptions about how it works. Each line tells us the cost of the underlying operation.
- *Pragmatic*—Code is concrete and solves today's problems. There are no unnecessary abstractions that can make the code harder to understand.
- *Testable*—Code is straightforward to test.

Achieving these qualities is not a walk in the park. The journey from learning to mastering Go can be quite the ride. To create idiomatic code in Go, you must understand the language mechanics. Otherwise, you'll fight with the language and bring your past knowledge from other programming languages to Go. Doing so will eventually bite and frustrate you.

Throughout this book, you'll discover insights and techniques to help you write high-quality idiomatic code in Go. What this book discusses will give you the proper mindset and set you on the right path to achieving your goals faster and reliably with Go.

Still, there is no single truth but guidelines.

Some people can look at the same code and think, *"Oh, this is terrible!"* while others may think, *"This is awesome!"* Ultimately, it's about creating code that can survive by quickly adapting to changing needs, which brings us to our next goal: writing testable code.

1.3 Crafting testable code

"Nothing endures but change."

— *Heraclitus*

Testable code has superpowers.

It's a shape-shifter, capable of adapting to new requirements and standing the test of time. However, software that resists change loses this crucial ability. With automated tests, you can confidently determine if our code works correctly even after modifications.

But even with tests, developing an entirely bug-free program is challenging. Our tests might have touched every line of the program, but they most likely didn't test every possible path. So, 100% test coverage doesn't mean our programs are bug-free.

Finding bugs is only one side of the testing story. Equally important is creating testable code—an art form in its own right. Testable code also tends to be easily adaptable. You can forge resilient, adaptable, and well-designed programs with testable code. When you write tests, you effectively put our code through its paces from the perspective of those tests. This allows us to experience firsthand how easy or challenging it is to interact with our code.

Lastly, tests can help us but should not be our end goal. Testing should be pragmatic, and you should not test just for the sake of testing. If you take testing too far, you might go to the extreme end of the testing spectrum. There is a thin line there. Luckily, we'll discover where that line starts and ends and explore pragmatic and reliable solutions.

Each chapter in the book is crafted to provide insights into the world of testable code. You may notice variations in the number of tests presented in different chapters. In Chapters 3 and 4, the emphasis is on testing itself to explain the fundamentals. As you progress in the book, discussions will shift toward writing idiomatic and inherently testable code. While the latter chapters may contain fewer tests, they dive into writing testable code.

Now that you know the book's goals, the rest of the chapter will focus on Go itself. Let's start with a short history of Go to understand why Go exists. Then, the chapter will briefly introduce Go's flagship features, such as concurrency and interfaces.

1.4 Why does Go exist?

In 2007, seasoned programmers came together to create Go:

- Ken Thompson (instrumental in the creation of the C language and UTF-8).
- Rob Pike (co-inventor of UTF-8).
- Robert Griesemer.

Russ Cox (now the tech lead of Go) and Ian Lance Taylor would join the team shortly after.

They aimed to address the issues they faced with other languages: slow compilation times and complex, cumbersome features in efficient languages like C, C++, and Java, as well as the lack of efficiency in developer-friendly languages like Python and JavaScript. They aimed to blend the best features of static and dynamic languages into a new one.

After five years of development, Go was released. It offered a solution that balanced simplicity, reliability, and efficiency in software development. This balance is a key reason for Go's success, appealing to developers and companies worldwide, including tech giants like Amazon, Apple, Google, and others, who now rely on Go for building modern software.

Note

The Go 1 backward compatibility ensures that code written ten years ago still runs with the latest release. For more details, see the link: https://go.dev/doc/go1compat

Trivia: Go vs. golang?

Rob Pike proposed the language's name in an email he sent on Sep 25th, 2007:

```
Subject: Re: prog lang discussion
From: Rob 'Commander' Pike
Date: Tue, Sep 25, 2007 at 3:12 PM
To: Robert Griesemer, Ken Thompson

I had a couple of thoughts on the drive home.

1. name
'go'. you can invent reasons for this name but it has nice pr
it's short, easy to type. tools: goc, gol, goa. if there's an
debugger/interpreter it could just be called 'go'. the suffix
...
```

Disney had already registered the domain "go.com". So, the Go team needed to register the domain golang.com instead, and the word "golang" was stuck in the language. To this day, most people call the language "golang". Of course, the actual name is Go, not golang.

But in a practical sense, the golang keyword makes it easier to find something related to Go on the web. You can read more about the history of Go at this link: https://commandcenter.blogspot.com/2017/09/go-ten-years-and-climbing.html

1.5 What is Go great at?

Go is great for crafting any program, but especially the following:

- *Client and Server Programs*—Go's standard library provides comprehensive networking support, including HTTP and TCP protocols, which simplifies the development of client and server programs such as microservices.
- *Distributed Network Programs*—The concurrency model in Go, centered around goroutines and channels, is a game-changer for writing distributed systems. It lets us manage multiple concurrent tasks without the overhead of traditional threading, making Go a perfect fit for

implementing distributed systems.

- *Command-line Tools*—Go's simplicity and its powerful standard library, makes it a great choice for developing cross-platform, efficient, and easy-to-deploy command-line tools.

Although Go is a general-purpose language, its largest user base consists of backend developers. Let's write a simple program to taste Go before exploring Go's big features.

1.6 Hello, gophers!

Let's start with a short and sweet program as a first step. This trivial executable program's goal is to say hello to gophers. As with every Go code, it's written in a package.

```
// hello/hello.go
package main                #A
import "fmt"                #B
func main() {
    message := "Hello Gophers 👋!"        #C
    fmt.Println(message)            #D
}
```

Note

Go programmers are conventionally called "gophers." See: https://go.dev/blog/gopher.

Running this program is straightforward. You can use the `go` tool to compile, link, and run the program. This tool is central to Go and comes with a compiler, linker, tester, etc.

```
$ go mod init github.com/username/repository    #A
$ go run .                    #B
Hello Gophers 👋!
```

Instead of getting bogged down in how this tiny program works, let's continue the tour. The next chapters extensively explain how to structure, design, and write Go programs. For instance, Chapter 2 explains how to create Go modules and packages from scratch.

Note

Go is a simple language and the language mechanics fit in a short document called Go Spec. Anyone can build a Go compiler by following the rules outlined in this document. For instance, TinyGo is one of them. Visit the link to see Go Spec, https://go.dev/ref/spec.

Go is a statically and strongly typed programming language

Go is a statically typed language (i.e., every variable's type is known at compile time). This enables the compiler to ensure type consistency and reduces the likelihood of trivial bugs.

Because the memory layout of the types is known at compile time, the compiler can generate optimized machine code. Additionally, code editors can do a better job at analyzing the code and help us while editing code. More importantly, types become the documentation.

Static typing has some potential drawbacks, too. Code is typically more verbose than dynamically typed languages. Luckily, Go fuses the ease of usage of dynamic languages in a statically typed language. For instance, in the example program, Go automatically guesses the `message` variable's type without having us provide the type as a `string` variable.

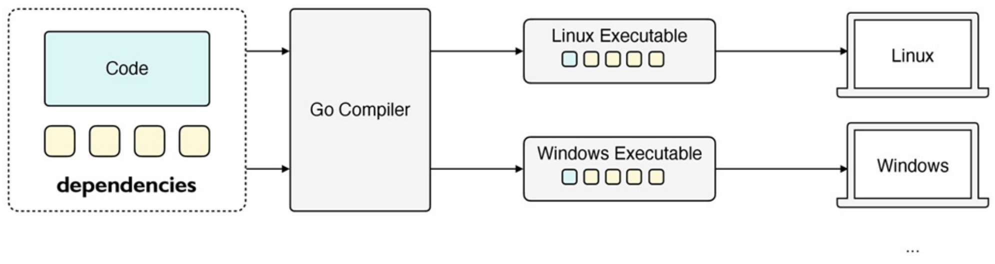
Note

Go is in a sweet spot between a dynamically typed language like Python and a statically typed language like C. It draws the best of power from both worlds. More on this soon.

1.6.1 Compilation and static binary

Go doesn't require an intermediary execution environment such as an interpreter or a virtual machine. As Figure 1.1 shows, Go compiles code into efficient (fast) native machine code targeted for major operating systems (e.g., Linux) and architectures (e.g., AMD64). Go compiles so fast that it feels as if you're working with an interpreted language like Python.

Figure 1.1 Go compiles code with all necessary dependencies into a single binary.



Go's approach differs from that of interpreted languages. Python, for example, requires a Python interpreter, while Java requires a virtual machine. In contrast, a Go binary only requires an operating system to run, making distributing Go programs straightforward.

The compiler produces a single *static binary* at the expense of a larger binary size. This downside dwarfs when compared to the advantages it brings. Imagine managing a fleet of a thousand Linux servers. All you need to do is compile the program once, copy it to the target machines, and run it without any concerns about dependencies.

Note

Chapter 6 dives into how to compile cross-platform binaries (Section 6.3).

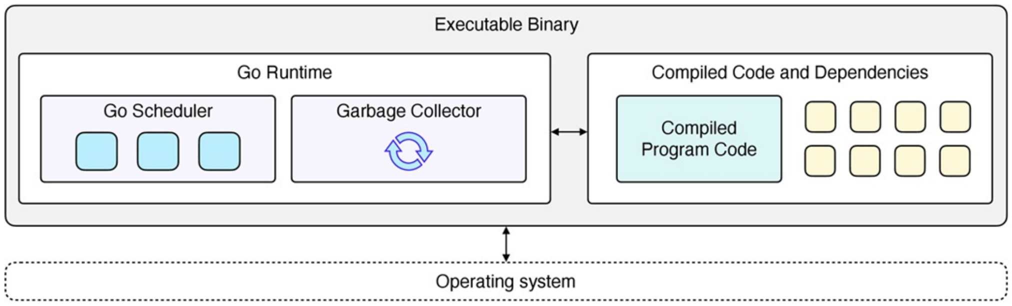
Dive deep: How does Go compile so fast?

Unlike Python or Java, Go compiles directly to efficient native machine code, bypassing an intermediary interpreter. Despite this compilation step, compiling a Go program is often quicker than most languages. Go's fast compiler helps us to develop rapidly, and its speed owes to its design. For more information, visit https://go.dev/talks/2012/splash.article

1.6.2 Go Runtime

Let's explore how Go achieves to run executable programs without any dependencies. In Go, every executable binary includes Go Runtime (which is automatically compiled for the target operating system). Figure 1.2 shows what's typically inside an executable Go program.

Figure 1.2 A statically linked binary that includes everything it needs to run.



An executable binary packs all its dependencies, including Go Runtime that has the following:

- *Go scheduler*—Orchestrates concurrent functions called *goroutines*.
- *Garbage collector*—The garbage collector runs in the background and automatically reclaims memory that is no longer referenced. Go's garbage collector is also concurrent and scheduled by the Go scheduler. If you're curious about the garbage collector's internals, visit https://golang.org/doc/gc-guide.

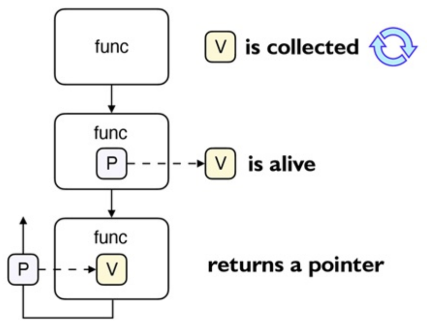
In Go, you can write code without worrying about freeing up unused memory.

1.6.3 Pointers

Unlike most other high-level languages, Go allows us to access memory with pointers directly, giving us explicit control. Unlike C and C++, thanks to Go's automatic memory management with a garbage collector, pointers are less dangerous and type-safe.

Thanks to the garbage collector, unlike C, in Go, it's perfectly fine for a function to return a pointer to one of its local variables. See Figure 1.3 for a rough illustration.

Figure 1.3 The garbage collector reclaims the memory space occupied by the variable (V) when there are no pointers (P) left referring to the variable.



There are three functions. The top function calls the middle one, which calls the bottom one. The middle function receives a pointer to the bottom function's local variable (V). Since the top function doesn't refer to the variable, the garbage collector reclaims the variable.

Let's now focus on Go's biggest features: concurrency and type system.

Note

Section 2.2 will further explain how pointers give us the ability to share variables across our program by referencing their memory addresses, rather than duplicating their values.

1.7 Concurrency

Go was born out of necessity while the language's creators worked with highly concurrent servers at Google. Go's basic idea of concurrent programming is that concurrent functions communicate by sending and receiving messages. This model draws inspiration from Tony Hoare's 1978 article "Communicating Sequential Processes." In Go terminology:

- "Sequential Processes" are goroutines that run a function concurrently.
- "Communication" between goroutines is enabled by *channels*.

Go abstracts classical concurrent programming constructs like threads and mutexes with goroutines and channels, simplifying concurrent programming.

Let's get an overall feeling of Go's approach to concurrency.

Note

Chapters 2 (Section 2.5), 7, and 8 go into detail for concurrent programming in Go.

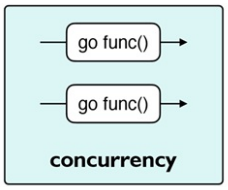
Note

See the Wikipedia link for details about Communicating Sequential Processes: https://wikipedia.org/wiki/Communicating_sequential_processes

1.7.1 Goroutines

Goroutines are independently running concurrent functions. Figure 1.4 is a simplified illustration of two goroutines that concurrently run an ordinary function separately.

Figure 1.4 Goroutines run ordinary functions concurrently. The go statement before a function spawns a new goroutine that runs the given function.



Concurrency is not an afterthought in Go; it's a fundamental part of the language and the Go Runtime. The beauty of Go is that it doesn't differentiate between concurrent or sequential code in terms of syntax. Writing concurrent code is as straightforward as writing sequential code. For example, say you have a function that crawls a web page content:

```
func crawl(url string) {
    ...
}
```

You can turn this function into a concurrent one like this:

```
go crawl("...")
go crawl("...")
```

This code concurrently runs the `crawl` function in the background by two separate goroutines. The `go` keyword turns a sequential function into a concurrent one.

In traditional programming languages, thread pools are often used to manage the high execution cost of running multiple threads. However, in Go, the story is different. Goroutines are incredibly lightweight, eliminating the need for a thread pool. This efficiency allows you to run millions of goroutines on a regular machine, opening up a world of possibilities.

How does it work? Let's dive in.

1.7.2 Go scheduler

An operating system runs executable programs in separate processes. Every process comprises smaller units called threads that run code on one of the CPU cores.

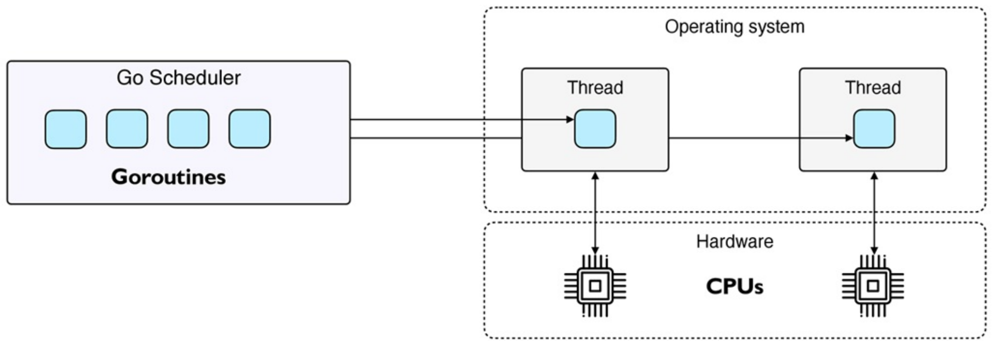
Switching between threads (known as *context switching*) is typically expensive: the operating system pauses and saves the state of the running thread, loads the state of the next thread, and runs the next one. This thread switching overhead can significantly degrade performance in programs with many threads competing for processor time.

Go's solutions to this problem are:

- Abstracting operating system threads with goroutines.
- Abstracting the operating system's scheduler with a goroutine scheduler.

As Figure 1.5 shows, the scheduler's job is to distribute goroutines to threads. There can be millions of goroutines waiting in the scheduler's queue to be scheduled on a thread.

Figure 1.5 The scheduler multiplexes goroutines onto threads for efficiency. Parallelism occurs at runtime when multiple CPU cores can run multiple threads in parallel (a nice side effect of structuring a program to run concurrently).



Context switching between goroutines is tremendously more efficient than thread context switching because Go's scheduler doesn't have to involve the operating system kernel (kernel-space calls are an order of magnitudes slower than user-space calls).

If a goroutine blocks (e.g., due to channel or I/O operations, etc.), the scheduler parks it, and another goroutine takes place on a free thread, reusing threads for goroutines. Also, goroutines are cheap and start their life with 2KB of stack memory that can grow or shrink as needed, significantly reducing the memory resource costs.

For I/O bound work (waiting for input/output operations like network calls to be completed), Go uses non-blocking I/O to avoid blocking underlying operating system threads. For instance, numerous goroutines can send thousands of HTTP requests with a single thread.

Go's approach minimizes the overhead of thread context switching, enabling Go to integrate concurrency into the language and lead to more efficient concurrent processing.

Note

Go scheduler's inner mechanics are more complicated than explained here. Check out the link if you're curious: https://youtu.be/watch?v=YHRO5WQGh0k

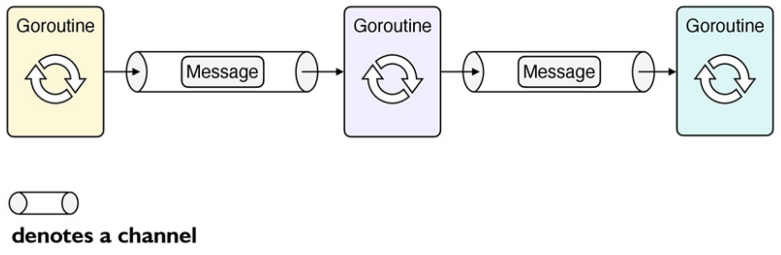
1.7.3 Channels

"Share memory by communicating, don't communicate by sharing memory."

— *Go Proverb*

Channels allow goroutines to communicate and synchronize with each other safely. Think of channels as network cables or Unix pipes between goroutines. Figure 1.6 illustrates an example of a few goroutines that communicate through channels.

Figure 1.6 Goroutines that are running concurrently communicate over channels. These goroutines forward the message to the next goroutine, each doing some computation on the received message before they send it to the next goroutine.



Here, you see the pipeline pattern where goroutines send messages through a channel to transfer the ownership of a piece of data to the next goroutine. While goroutines execute, sending and receiving a message pauses them for a while so they can *synchronize*.

While using classical concurrency primitives like mutexes (which Go supports) is fine for serializing and protecting concurrent access to a shared resource (e.g., a counter or a cache), the idiomatic way is to use channels when you want goroutines to communicate (passing ownership of a piece of data, distributing tasks, communicating results).

When structured correctly, this message-passing approach simplifies concurrent programming by making the data flow explicit and reducing the likelihood of concurrency-related bugs. The book dives into how to structure idiomatic and testable concurrent programs correctly, starting from Chapter 7 (also see Section 2.5 for an intro).

1.7.4 Wrap up

"If all you have is a hammer, everything looks like a nail."

—Abraham Maslow's law of the instrument.

Many newcomers to the Go programming language often fall into the trap of applying Go's concurrency features to every problem they encounter. This is mainly because Go makes it relatively easy to turn sequential programs into concurrent ones.

However, this approach is rarely the best one.

While Go's concurrency model makes it easier to work with concurrent code, humans are naturally wired to understand sequential execution better. Therefore, it's important to resist the temptation of using concurrency unless it's necessary for solving the problem at hand.

1.8 Type system

"If you could do Java over again, what would you change?" "I'd leave out classes," he replied. ...The real problem wasn't classes per se but rather ... inheritance.

— James Gosling (Java's inventor)

The next standout feature of Go is its simple and effective type system.

Unlike other popular programming languages that rely heavily on classes and inheritance, Go implements a lean version of object-oriented programming. Go's type system is flat, meaning that there are no type hierarchies, classes, or inheritance. Polymorphism is only possible via interfaces. Go favors composition over inheritance.

While it is possible to imitate similar flat types in other languages by avoiding inheritance and using only interfaces to achieve polymorphism, Go makes it a requirement. This approach prevents unintentional damage caused by overusing type hierarchies.

Now, let's take a high-level overview of embedding and polymorphism in Go.

It's okay if you don't understand the examples yet, as the later chapter will delve into these concepts and explain them in detail. Nonetheless, having a rough understanding is sufficient for now.

1.8.1 Embedding instead of inheritance

Go's type system eschews classical inheritance in favor of a flat structure.

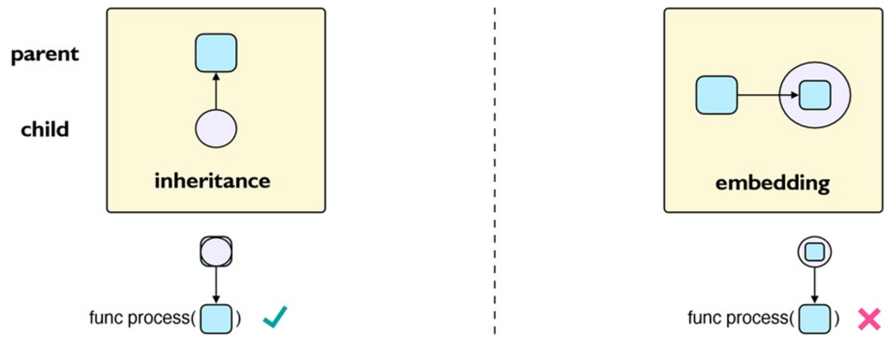
Instead, Go has struct embedding, allowing one struct type to include another's data and methods, akin to composition. However, unlike inheritance, embedding doesn't establish a parent-child relationship; the types remain distinct and aren't interchangeable.

Note

Think of a struct type similar to a lightweight class.

Figure 1.7 illustrates the differences between inheritance and embedding.

Figure 1.7 While inheritance creates a type hierarchy, embedding does not. In inheritance, you can pass parent (square) and child (circle) types interchangeably to a function that expects only the parent type. With embedding, the function can only work with the specified type. Embedding does not form a type hierarchy. Embedding simply makes one type's methods and data part of another type.



The distinction is crucial: when a struct embeds another, it doesn't inherit from it. This concept helps maintain clear boundaries between types and can prevent some of the common pitfalls of inheritance, like tight coupling and difficulty in understanding code hierarchies. It's part of what gives Go its characteristic simplicity and power in design.

Programmers new to Go are often surprised when they see they can't use a struct type that embeds another in place of the embedding struct type. They are surprised because they mistakenly think that embedding is inheritance, which Go does not support.

Note

Chapter 2 (Section 2.4.1) explains struct types, and Chapter 9 (Section 9.3.1) explores embedding in more detail.

1.8.2 Implicit interfaces

Go interfaces blend the flexibility of dynamically and statically typed languages. You can pass types to functions based on their methods (duck typing). At the same time, Go brings static typing benefits; like Java, Go checks at compile time that the types match the interface's methods. Go's approach provides both flexibility and reliability in your code.

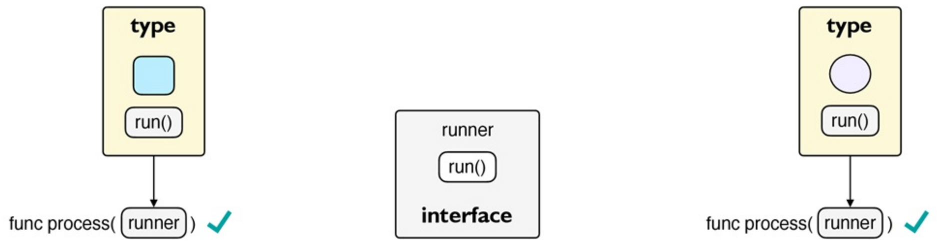
Go can achieve this flexibility because interfaces are implicitly satisfied. Types do not have to explicitly mention that they implement an interface. If a type implements all the methods of an interface, you can use the type wherever that interface is expected (duck typing: if it quacks like a duck and walks like a duck, then it should be a duck).

Note

Go's implicit interfaces empower you to craft code that's easy to manage and adapt.

Figure 1.8 illustrates an example usage of Go's implicit interfaces. Say, you have an interface called `runner` with a `run` method, the square and circle types, each with a `run` method (behavior), and lastly, a function that expects the `runner` interface.

Figure 1.8 The function expects a type that implements the `runner` interface (any type with a `run()` method). You can interchangeably pass either of these types to the function because each has a `run()` method and implicitly satisfies `runner`.



Notice that these types say nothing about the `runner` interface. Each implements only the `run` method, which allows you to pass any of them to the `process` function. If you pass a type without the `run` method to the function, the code won't compile.

In practice, Go's implicit interfaces offer an unmatched level of simplicity, flexibility, and reliability in program design. By decoupling the concrete implementations from their behaviors, Go encourages a more modular and scalable code structure.

You don't have to design interfaces upfront. Go's approach empowers you to write code that is easy to maintain and adaptable to future changes without requiring extensive refactoring.

Note

Chapter 2 dives into methods and interfaces (Sections 2.4.2 and 2.4.3). Most of this book is about the idiomatic usage of methods and interfaces.

Note

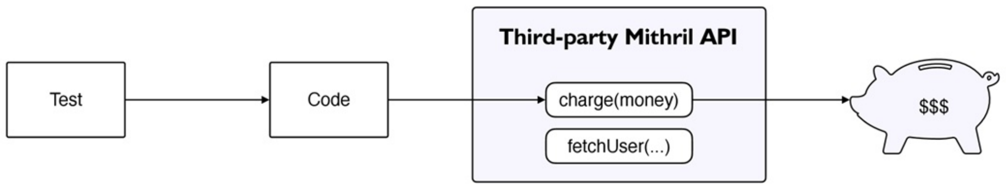
Go also has limited support for generic type parameters. See the link for more information, https://go.dev/doc/tutorial/generics. Section 2.4.4 also explains generics.

Decoupling and testing with implicit interfaces

Let's briefly explore how Go's implicit interfaces boost the testability of code.

Mithril is an imaginary payment processor, and you want to test code that uses Mithril's API. As Figure 1.9 shows, when you run the test, the code under the test charges users for money! How can you test this code without sending requests to Mihtril's live servers?

Figure 1.9 This expensive test runs code that invokes the charge method of Mithril's API and contacts live servers. It's not the best way of testing.

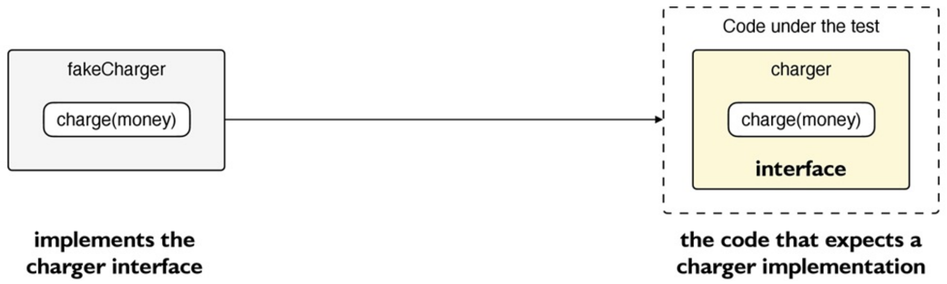


The problem is Mithril's API doesn't offer interfaces, which poses a challenge for testing. This is where Go's implicit interfaces shine. Instead of costly tests against Mithril's servers, you can decouple your code by declaring a tiny interface with a subset of Mithril's API methods.

Here's the recipe (see also Figure 1.10):

1. Declare an interface (i.e., `charger`) with a single `charge` method.
2. Create a test-only fake payment processor type with a `charge` method.
3. Change the code to accept the `charger` interface.
4. Pass this fake type to the code while testing.

Figure 1.10 Testing the code with a fake payment provider. The code no longer uses Mithril's API methods. Instead, it uses the fake payment provider.


fakeCharger
charge(money)
implements the charger interface
Code under the test
charger
charge(money)
interface
the code that expects a charger implementation

The code no longer depends on Mithril's API; instead, it accepts the `charger` interface. This approach enables you to pass any type with a `charge` method to the code (including the fake payment processor while testing or the actual Mithril's API). Mithril's absence of an interface was never an obstacle for testing. In Go, you can always defer declaring an interface until needed, making your code straightforward to evolve and also testable.

This is one of the prime examples of Go's practicality. Go allows you to craft testable code without designing types upfront, effectively allowing you to evolve your code with confidence, even when third-party interfaces aren't available. This technique, along with others you'll learn in this book, showcases how Go is engineered for building solid software.

Note

Chapter 6 and Chapter 11 (Section 11.5) dive deeper into testing with interfaces.

1.9 Standard Library

Go comes with a rich set of packages called the standard library. These packages include functionality, from simple tasks like string manipulation to running HTTP servers. By reusing the functionality offered by the standard library, you can develop your programs rapidly.

For instance, here is an example of running an HTTP server using the standard library:

```
package main
import "net/http"
func main() {
    handler := func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("hello, gophers!"))
    }
    http.ListenAndServe("localhost:8080", http.HandlerFunc(handle
}
```

This program listens on port 8080 of localhost. Once it receives an HTTP request, the server sends a "hello, gophers!" message to the client. Thanks to concurrently serving requests with goroutines, even this basic server can handle tens of thousands of requests per second.

Note

Chapter 9 explains how to structure and write HTTP servers in detail.

Besides the standard library, Go has a vibrant and mature ecosystem of mature third-party packages. It's also straightforward to download and use these packages using Go's built-in module system. Yet, it's wise to use the standard library if it offers the packages you need.

The standard library is a tribute to reliability, having undergone rigorous testing and tight integration with the Go runtime. This often leads to more robust and efficient code. Moreover, most Go developers are familiar with the standard library code, which means it's easier for everyone to understand each other's code and work together effectively.

Note

You can see the standard library packages at the link: https://pkg.go.dev/std.

This book uses the standard library for all the examples in the book. For example, Chapter 3 introduces the `testing` package for automated testing. Chapter 5 uses the `flag` package for command-line flags. Chapter 8 uses the `http` package for sending requests, and so on.

1.10 Tooling

Another reason Go is so successful and popular is its great built-in tooling support. Go tools play a critical role in a gopher's workflow, from automatically managing dependencies and formatting code to compiling. You can use all the tools through a single `go` command.

For example:

```
$ go test .
PASS
```

Go prioritizes tests as first-class, and with the `go test` tool, you can effortlessly run tests, benchmark, and profile your code to identify areas where it's spending the most time. Or, you can also use the `go fmt` tool to format your code automatically.

From this:

```
package main;import "fmt"
func main() {    fmt.Println("hello, gophers!" )}
To this:
package main

import "fmt"

func main() {
    fmt.Println("hello, gophers!")
}
```

The formatting tool is tremendously helpful for keeping code consistent across every Go project. No more tabs vs. space wars. Here's a list of a few more tools:

- `go build`: Compiles Go code.
- `go run`: Compiles and runs Go code.
- `go get`: Downloads third-party packages and add them to your project's Go module so that you can import them into your program.
- `go vet`: Finds potential issues in Go code to improve code quality.
- `go doc`: Displays documentation for packages.
- `go generate`: Generates Go code for common tasks.

These are just a few examples of the many built-in tools in Go. Thanks to Go's tooling, you can focus on coding. Visit the link for other Go tools: https://pkg.go.dev/cmd/go.

1.11 Downloading the book's source code

If you wish to follow along with the examples provided in this book (which I strongly recommend), you may want to access the corresponding source code. To make things easier for you, the source code for examples is available from two different sources.

- https://github.com/inancgumus/gobyexample
- https://www.manning.com/books/go-by-example

Before getting started, you also might want to install the latest version of Go:

- https://go.dev/doc/install

This chapter has dissected why Go exists and the problems it's crafted to solve, cutting through the trivia and diving into the heart of what makes Go stand out—its simplicity, pragmatism, and efficiency. With an introduction to Go's strengths, you've begun the journey to understand how Go makes it easier to construct reliable and performant software.

As the first chapter concludes, I hope you're beginning to recognize the reasons to delve further into the world of Go and continue reading this book. This book is structured to help you develop the right mindset for utilizing Go language's idioms and mechanics effectively. It will serve as your guide in crafting idiomatic, maintainable, and testable code in Go.

1.12 Summary

- This book will be your guide to writing idiomatic and testable code.
- Learning through examples is an effective way to learn a new technology.
- Go is a simple and effective programming language.
- Go fuses the best sides of dynamically and statically typed languages.
- Go code is written in packages and compiled into efficient native machine code.
- The compiler produces a single static binary, making deployment straightforward.
- Every executable comes with Go Runtime, including a scheduler for concurrent execution and a garbage collector to automatically reclaim unused memory.
- Go doesn't hide the cost of doing something and has pointers to use memory directly. Pointers enable us to share values without copying them. The garbage collector can automatically free unused memory referred to by the pointers.
- Automatic memory management allows returning pointers from functions.
- Concurrency is built-in in the Go language. There is no difference between writing sequential and concurrent code. It is straightforward to turn the prior to the latter.

- The Go scheduler abstracts the operating system scheduler and threads.
- Goroutines are independently running concurrent functions.
- Channels allow goroutines to communicate and synchronize with each other.
- Go's type system is flat and does not support classical inheritance.
- Embedding allows a type to be embedded into a struct type.
- Embedded and embedded types are different and don't form a type hierarchy.
- Interfaces are explicitly satisfied, decoupling implementations from interfaces.

2 Go Crash Course

This chapter covers

- Packages and modules.
- Variables and pointers.
- Collection types: arrays, slices, and maps.
- Object-oriented programming.
- Concurrent programming.
- Error handling.

This chapter is a fast-paced primer for experienced programmers to prepare for the more advanced content we cover in the book. While other chapters focus on idiomatic Go and testability, this chapter focuses on the basics of Go's distinctive features and mechanics.

Treat this chapter as a warm-up rather than a mastery. Use it as reference material and return to it as you read the book. And don't worry if some of the content in this chapter is challenging to understand, as it'll become clearer as you progress in the book.

Note

The source code of this chapter is at the link:
https://github.com/inancgumus/gobyexample/tree/main/basics

2.1 Packages

We'll start with a simple executable program example that prints this book's title.

This is our program's directory structure:

```
.                 -> The project's root directory (new directory)
├── go.mod        -> Defines our project's name and dependencies
├── hello.go     -> Implements the package main
└── book         -> The book package's directory
    └── book.go  -> Implements the book package
```

Let's create a new directory on our system. We can create the rest later.

2.1.1 Overview

In Go, we write our code inside packages. There are two kinds of packages:

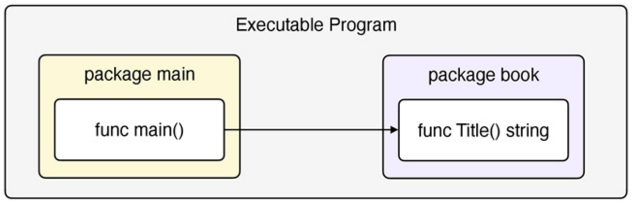
- Packages named `main` are for building executables and cannot be imported.
- Packages named anything but `main` can be imported from other packages.

Definition

A package is a set of Go source code files in the same directory.

We'll have two packages for our example program: `main` and `book`. See Figure 2.1.

Figure 2.1 The main package imports the book package to use its functionalities.



We'll start with the importable `book` package. The `main` package we'll write after will import the `book` package and call the `Title` function to write this book's title to the console.

2.1.2 Importable packages

As the next listing shows, the `book` package declares two functions: `Title` and `subtitle`. They don't take an input parameter but return a string. The `package` statement makes the `book.go` file a part of the `book` package. There can be many files inside a package.

Listing 2.1 The book package (book/book.go)

```
// Package book offers information about the Go by Example book.
package book                #A

// Title returns the title of this book.
func Title() string {           #B
    return "Go by Example: "+ subtitle()
}

func subtitle() string {           #C
    return "Programmer's Guide to Idiomatic and Testable Code"
}
```

We've created an importable package named `book`. Let's talk about *exporting*.

Note

We can export a package's items by capitalizing their first letter to make them public.

Go relies on packages as its sole isolation mechanism.

The code in the same package can access everything within their package. However, other packages can only access the items a package exports. For instance, the `main` package can import the `book` package and call `Title` because it's exported, but not `subtitle`. Still, `Title` can access `subtitle` because they are within the same `book` package.

`Title` as a function name is idiomatic, while `GetTitle`, `GetBookTitle`, or `BookTitle` are not. Is the book itself getting the title? Of course not. Prefixing `Title` with "Get" adds unnecessary noise. Omitting "Get" is also crucial for readability when calling `Title` from other packages: `book.Title()`, precisely meaning the book's title. We also don't want to repeat (*stutter*) the package name, as `book.BookTitle()`. Again, unnecessary noise.

Note

Every source file in a package directory must declare the same package. Idiomatic package names are nouns, descriptive yet concise, like `book`. They are lowercase and lack underscores or mixed caps. Ideal names convey what a package offers.

Dive deep: Test packages

Tests can declare a test package, like `book_test,` in the same directory with the `book` package because they're only included while testing—we'll cover testing later.

2.1.3 Package main

In the next listing, the `main` package imports the `book` package and calls `Title`. Then, `main` passes the returned title to the `fmt` package's `Println` function to print the title.

Listing 2.2 Implementing package main (hello.go)

```
package main


import (
    "fmt"
    "github.com/inancgumus/gobyexample/book"
)

func main() {
    fmt.Println(book.Title())
}
```

While importing packages, the last segment—`book`—in the import path is the package's name. Go Standard Library packages, like `fmt`, don't need a full path.

Note

Defining a package as `main` signals to the Go toolchain that it contains the program's entry point. The `main` package doesn't automatically make the package executable—the `main` function within the `main` package signals that the package should be built as an executable binary and allows designating the `main` function as the program's entry point.

Running the program

Now that we have all the parts of our program, it's time to run it. We can use the `go run` command to compile and run our executable program. The `main` function is an executable program's entry point and it'll be called when we run our program.

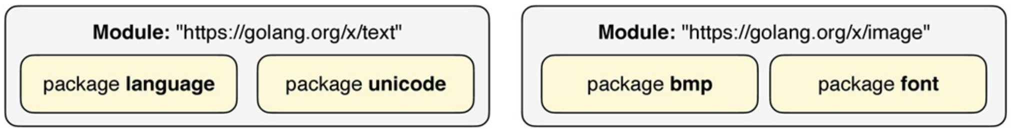
```
$ go run .
go: go.mod file not found in current directory or any parent dire
```

We ran our program but received an error. Let's fix this error by initializing a module.

2.1.4 Modules

For every new project, we must initialize a module to manage dependencies. As shown in Figure 2.2, conceptually, a module is a collection of packages with a unique name, often a URL, to ensure uniqueness. In practice, a module is simply a file named `go.mod` in a project's root. It's like `package.json` in Node.js or `requirements.txt` in Python.

Figure 2.2 A module is a collection of packages.


Module: "https://golang.org/x/text"
package language
package unicode
Module: "https://golang.org/x/image"
package bmp
package font

Note

While we can use non-URL module names, this might lead to difficulties in locating and downloading the module. Using a URL ensures that our modules can be uniquely located.

Let's go to our project's directory and initialize a module with the following command.

```
$ go mod init github.com/inancgumus/gobyexample    #A
```

This command creates a `go.mod` file in the current directory with the following content.

```
module github.com/inancgumus/gobyexample    #A
go 1.22                          #B
```

This file can include the module dependencies of our module. For now, we don't depend on other modules. We'll see how to download and use other modules in later chapters.

Note

Throughout the book, we'll use the same module we just initialized. Visit the link to learn more about Go modules: https://go.dev/blog/using-go-modules.

2.1.5 Building and running

Now that we initialized the module, we can finally compile and run our program.

```
$ go run .    #A

Go by Example: Programmers Guide to Idiomatic and Testable Code
```

The `run` command is excellent for prototyping and throws away the executable binary after the program stops. The `build` command lets us keep the executable:

```
$ go build -o hello    #A

$ hello        #B
Go by Example: Programmers Guide to Idiomatic and Testable Code
```

Deployment of this binary is simple since it's self-sufficient without any external dependencies—*build, copy, and run it on a target machine*. Still, we must recompile the code for target platforms. Thankfully, Go can compile to major operating systems, including Linux, Windows, and macOS, and a wide range of architectures, including AMD64, ARM, MIPS, WASM, and more.

We'll look at how to cross-compile in Chapter 6 (Section 6.3).

2.1.6 Exercises

- Create a new directory and initialize a Go module.
- Create an importable package and import it from the `main` package. Your package should have exported and unexported functions that you can call from `main`.
- Compile and run your program using `go build` and `go run`.

2.1.7 Wrap up

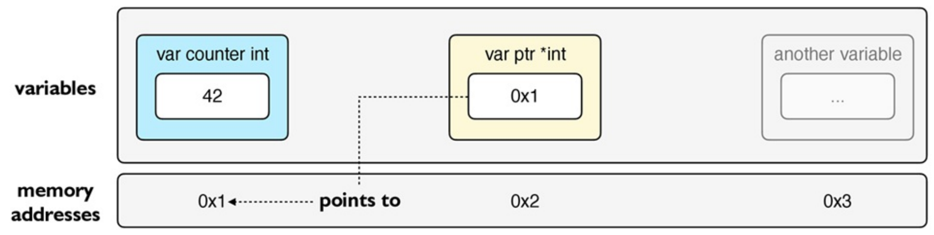
- The `main` package's `main` function is an executable program's entry point.
- Packages named other than `main` can be imported by other packages.
- Packages can only access the exported items of other packages.
- Exporting is done by capitalizing the first letter of an item inside a package.
- The `go mod init` command initializes a Go module.
- The `go run` command runs an executable program and throws away the binary.
- The `go build` command compiles packages.
- Go Runtime is included with every executable Go program and contains a Go scheduler to orchestrate goroutines and a garbage collector to recycle memory.

It's time to focus on Go's distinctive features so we can be ready for the next chapters. We'll start with pointers, crucial for being effective at Go. Then, we'll dive into collection types like arrays and slices, object-oriented programming, concurrency, and lastly, error handling.

2.2 Pointers

Every variable stores a value at a unique spot in computer memory tagged with a numeric address (typically depicted as hexadecimal). While some of these variables store raw values, others store memory addresses of other variables. A pointer is a variable that stores another variable's memory address. See Figure 2.3 to see how variables are stored in memory.

Figure 2.3 A variable stores a value at a memory location tagged with an address. The numbers below are simplified memory addresses of each variable shown.



We can declare these variables like this:

```
counter := 42        #A
ptr     := &counter     #B
```

The `counter` variable stores an integer value of `42` in the memory spot tagged with an address of `0x1`. While the `ptr` variable is a pointer at address `0x3`. This pointer stores the first variable's memory address of `0x1`—it's *pointing to* `counter`. Keep in mind that these memory addresses are for illustrative purposes. They will be different at runtime.

For those from programming backgrounds without pointers, think of a pointer as the book index on the back cover. It tells us where the page is rather than the page's content. Otherwise, the book index would copy the page's content instead of referring to it.

```
index (pointer)                  page content (data)

Look at page 42 for pointers -> Pointers are ...
```

Or a URL to a web page is a pointer to that page. The URL doesn't contain the page content, but it points to where we can find the information. It's a rough analogy but it might help.

```
url (pointer)                    web page content (data)

http://foo.com/abc            -> lorem ipsum dolor sit amet...
```

If you're already familiar with pointers from languages like C or C++, Go's pointers will feel similar, but Go disallows pointer arithmetic, and it uses a garbage collector. For instance, in C/C++, returning a pointer to a local variable *might be* a bug that can lead to surprising behavior. In Go, the garbage collector keeps track of pointers and the values they reference, so it's safe to return a pointer from a function; the memory won't be reclaimed until no references are left. We'll cover these differences in detail throughout the book.

2.2.1 Variables and pointers

Let's look at variables and pointers in more detail.

The following declares an `int` (integer) variable named `counter` without an initial value.

```
var counter int
```

Note

Go is statically typed, and we cannot change variable types after declaring them.

Let's also declare another variable. This time, a pointer variable. Its name is `ptr`, and type is `*int` (a pointer to an `int` variable). Its value is `nil` (doesn't point to a memory address).

```
var ptr *int
```

Note

Pointers are type-safe. For instance, we can only assign an int variable's memory address to the pointer variable ptr. This provides type safety and makes security breaches less likely.

Lastly, let's declare a variable that uses the `Duration` type from the `time` package.

```
import "time"
```

```
...
var responseTime time.Duration
```

In Go, every variable is always initialized to a *zero value*. This reduces the chances of certain memory corruption bugs that can crash or open a program to attacks.

```
fmt.Println(counter)       // prints 0
fmt.Println(ptr)           // prints nil
fmt.Println(responseTime)  // prints 0s    #A
```

Note

Every variable declared without an initial value gets a zero value. For numbers, the zero value is 0; for strings, it's an empty string—""; for booleans, it's false; for pointers types, it's nil.

Now, let's start over and look into *type inference*. This time, we'll declare the variable `counter` using *the short declaration syntax* and give our variable an initial value.

```
counter := 42        // initialized with 42
fmt.Println(counter) // prints 42
```

The compiler infers the `counter`'s type as `int` because we initialize the variable with an `int` *constant* of 42. Since Go is type-safe, we can now assign only an `int` value to `counter`.

```
counter  = 42        // ok: assigning an int value
```

But the following won't work and result in a compile-time error.

```
counter  = int64(84) // error: 84 is int64, not int

counter  = 3.14      // error: 3.14 is float64    #A
counter  = int(3.14) // ok: counter is 3
```

We needed to *convert* 3.14 to an `int` value of 3, losing the fractional part. Now that we looked at variables, let's dive into pointers and address operators.

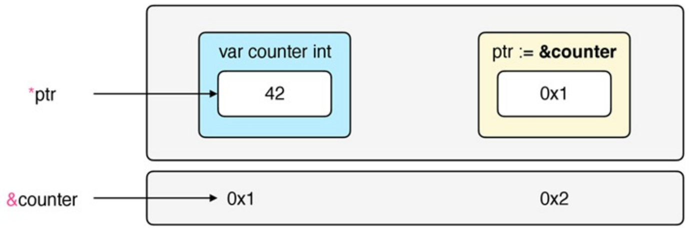
Note

Remember not to confuse `:=` with `=`. The former is a short declaration; the latter is an assignment. Here is more information: https://stackoverflow.com/a/45654233.

2.2.2 Address operators

As discussed, a pointer is a variable that stores the memory address of another variable. Figure 2.4 shows two variables: `counter` (an `int`) and `ptr` (an `*int`—a pointer to an `int` variable). The latter is a pointer and stores the `counter` variable's memory address.

Figure 2.4 Declaring a pointer that references a variable's memory address and dereferencing the pointer to find the value inside that variable via the pointer.



Similar to C, we use the following operators to utilize pointers:

- The ampersand "`&`" before a variable returns the variable's memory address.
- The asterisk "`*`" before a pointer returns a variable's value pointed by the pointer.

Using these operators, we fetch and put the `counter`'s memory address (`0x1`) into the pointer variable, `ptr`. Then, we indirectly fetch the `counter`'s value (`42`) through `ptr`.

Let's look into this in practice. We'll first declare a `nil` pointer variable. Like `NULL` or `null` in other languages, `nil` means a pointer that points nowhere. It's the zero value for pointers.

```
var ptr *int            // this variable's type is *int (a pointe
fmt.Println(ptr == nil) // prints true (doesn't point to an addre
```

Note

We can compare only pointers to `nil`. Types like slices, maps, interfaces, channels, etc. implicitly involve pointers and we can compare them to `nil`. More on this later.

Like the figure, let's declare `counter`, take, and store its address in `ptr`.

```
counter := 42        #A
ptr      = &counter    #B
fmt.Println(ptr)        // prints 0x1 (assuming that's the counte
```

Since `ptr` points to `counter`, we can fetch `counter`'s value by dereferencing `ptr` using `*`.

```
fmt.Println(*ptr)       // prints 42 (counter's value)
```

Note

Don't confuse dereferencing with a pointer type. `*ptr` is dereferencing since the star is before a pointer, whereas `*int` is a pointer type since the star comes before a type.

We dereferenced a pointer that points to a memory address. If we dereference a `nil` pointer, our program will panic (crash). That's why we should be careful while working with pointers.

```
ptr = nil               // now, the pointer doesn't point anywher

fmt.Println(*ptr)       // panics (crashes in runtime)
```

To sum up, `&` gets a variable's memory address, and `*` gets a variable's value through a pointer. Now, let's dive into pass-by-value to see where pointers might be helpful.

Dive deep: Unsafe

Because Go has a garbage collector, it doesn't allow pointer arithmetic. It's a design choice in favor of a more secure approach. Still, using the `unsafe` package, we can do pointer arithmetic. Although we rarely use `unsafe` since it defeats the purpose of the type system and safety, it might become useful. This article discusses using CGO and `unsafe` for talking to C code: https://go.dev/blog/cgo.

2.2.3 Pass by value mechanics

Go is pass-by-value. Passing a variable as an argument to a function copies that variable's value into a *local variable* within the function. Any changes to that variable will stay local to the function and won't be visible outside of it. Let's look at the next listing for an example.

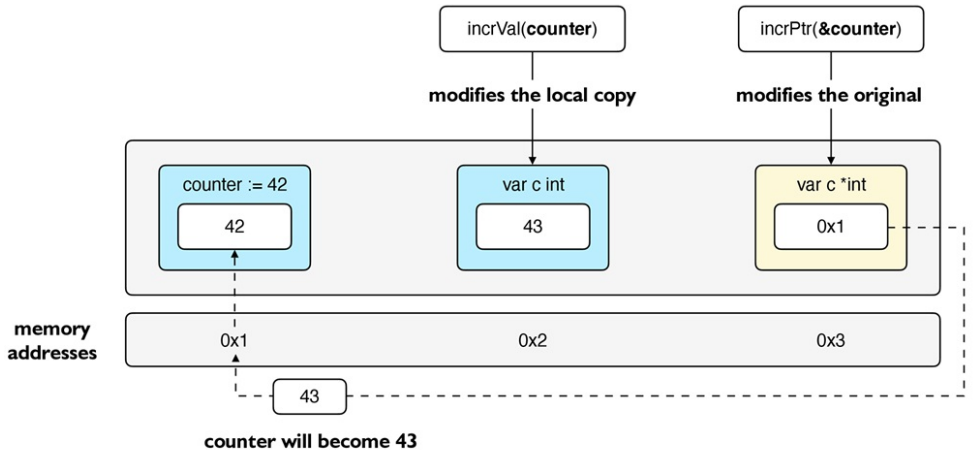
Listing 2.3 Passing a pointer to a function (ptr/ptr.go)

```
package main

import "fmt"

func main() {
    counter := 42

    incrVal(counter)            #A
    fmt.Println(counter) // 42

    incrPtr(&counter)        #B
    fmt.Println(counter) // 43
}

func incrVal(c int)  {
    c++
}

func incrPtr(c *int) {
    *c++                #C
}
```

We have three functions: `main`, `incrVal`, and `incrPtr`. Inside `main`, we declare a variable named `counter` with a value of 42. Then, we try to increment the variable with the `incrVal` and `incrPtr` functions. While the first one fails to do so, yet the second one succeeds.

See Figure 2.5 to find out why.

Figure 2.5 Pointers enable us to mutate original values without copying them. Numbers below are the memory addresses of each variable.



The short answer is `incrVal` takes a *value type*—`int`—while `incrPtr` takes a *pointer type*—`*int`, which is a pointer to an `int` value. Here's the long answer.

- Calling `incrVal` copies `counter`'s value to its local variable `c` of type `int`. Mutating `c` doesn't modify the original: `counter` remains 42 while the local one becomes 43.
- Calling `incrPtr` copies `counter`'s address to `c` of type `*int`. It finds the `counter`'s address and updates the `counter`'s value to 43.

In summary, after passing them pointers, functions can update the original value. The key to remember here is that pointers are copied, too. They just point to the same original variable.

Note

There can be many pointers pointing to the same value. For instance, we could declare `ptr2 := &counter` and it would point to the `counter` variable as `ptr` does.

Dive deep: stack and heap memory

In Go, functions can return a pointer to a function-local variable. For instance:

```
func newInt() *int {
    n := 42
    return &n    #A
}
```

The `newInt` function returns a pointer to its local `int` variable `n`. Although we rarely return a pointer to basic types like an `int`, returning pointers from functions is fine. For instance, a function can print `42` via the returned pointer after calling `newString`:

```
func main() {
    m := newInt()
    fmt.Println(*m) // prints 42
}
```

Once `newInt` returns a pointer to its local variable, `n` typically moves from the stack memory to the heap memory ("escapes to heap"). So, while local variables like `n` usually reside in the stack memory, returning a pointer might allocate the variable on the heap memory (but it's not a guarantee, the compiler might decide otherwise). The `main` function can access `n`'s value because the heap memory is shared between functions (unlike the stack memory).

Unlike stack memory, heap memory can become fragmented over time and filled with garbage values that are no longer needed. This is where Go's garbage collector steps in. It automatically watches heap memory and periodically reclaims unreferenced variables.

We don't need to concern ourselves with these details as Go automatically handles memory management. However, understanding this can help for a deeper understanding of Go.

2.2.4 Exercises

- Declare a few variables. Try assigning different types of values to every variable you create and observe the compiler errors. Increment integer

variables. Pass and mutate these variables in a function and check if your changes are visible outside.
- Declare a pointer variable to one of these variables and pass them to functions. Mutate the variables through the pointers and observe your changes from the caller.

Tip

You can find the basic types at the link: https://go.dev/ref/spec#Types

2.2.5 Wrap up

Pointers are so helpful that many languages *implicitly* use them. Take Java, for instance: every object is essentially a pointer. But Go, like C, is candid about pointers, making them *explicit*, leaving the decision of using a pointer to us for precise control over memory. We'll later dive into when to use a pointer and when not to. Let's wrap up.

- Variables are strongly typed, always initialized, and stored in a unique memory spot.
- Pointers are type-safe and store the memory addresses of other variables.
- Go is pass-by-value, and values are always copied when passing them around.
- Pointers let us work with original values without copying them.

Dive deep: Some types implicitly involve pointers

Although we say that the handling of pointers in Go is explicit, certain types implicitly involve pointers because of necessity or to make things easier. For instance, slices (dynamic arrays) have a pointer to their underlying arrays. Maps are pointers to complex data structures in Go Runtime. Similarly, interfaces, function values, and channels use pointers.

We'll explore these topics in this chapter and continue throughout the book.

2.3 Collections

Go has three built-in collection types: arrays, slices, and maps.

- Arrays are fixed-length collections of elements of the same type.
- Slices are like dynamic arrays that can expand and shrink.
- Maps are key-value pairs to access elements through their keys in constant time.

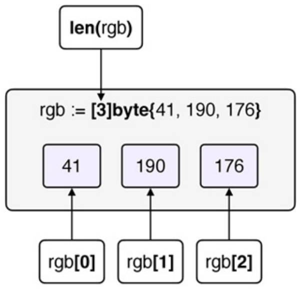
2.3.1 Arrays

Say we want to store RGB (red, green, and blue) colors in an array:

```
rgb := [3]byte{41, 190, 176}  #A
```

Figure 2.6 illustrates the `rgb` array and the operations we can use on it.

Figure 2.6 A byte array with three consecutive byte elements laid out in memory. While the built-in `len` function returns the array's length, 0-based indexing returns an array element at that index (e.g., `rgb[0]` returns the first element).



Note

The `byte` type is an alias for the `uint8` type (an unsigned 8-bit integer). We can also declare arrays of any length and type, such as `[10]string`, `[16]bool`, etc.

This array's type is `[3]byte` (an array of three bytes). An array's length and element type determine its type (which cannot be changed without recompiling the program).

It might be surprising, but arrays with different lengths have different types:

```
rgb = [4]byte{214, 65, 79, 255}    #A
```

Passing arrays to functions

Now, say we want to invert the colors of an 8K image (7680 x 4320 pixels). Since each pixel has RGB colors, we declare an array with ~100 million bytes (7680 * 4320 * 3). Then, we pass the array to a function that loops over and mutates each array element.

Listing 2.4 Passing arrays to functions (arrays1/array.go)

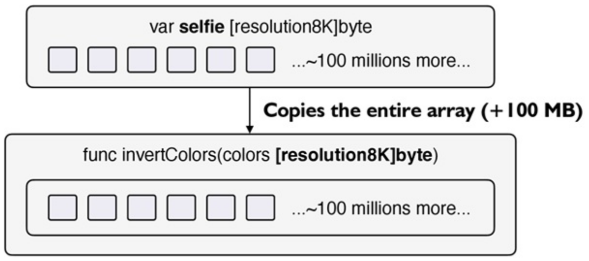
```
package main

const resolution8K = 7_680 * 4_320 * 3            #A

func main() {
    selfie := [resolution8K]byte{ /* ..color data.. */ }    #B
    invertColors(selfie)
}

func invertColors(colors [resolution8K]byte) {        #C
    for i := range colors {                #D
        colors[i] = 255 - colors[i]            #E
    }
}
```

We use a constant (for readability) to declare an array that occupies 100 megabytes of memory space. The invertColors function takes the array and loops over each element. The index variable i gets updated from 0 to the array's length - 1 (len(colors)-1). Inside the loop, we mutate each array element to invert the colors. So far, so good. However...

Passing pointers to functions

Passing the array to `invertColors` copies ~100 megabytes of data from one spot in memory to another, which is inefficient. As Figure 2.7 shows, the function gets a copy of the array.

Figure 2.7 Passing an array to a function copies the entire array.


var **selfie** [resolution8K]byte
...~100 millions more...
Copies the entire array (+100 MB)
func invertColors(colors **[resolution8K]byte**)
...~100 millions more...

Instead, we can use pointers for efficiency. As the next listing shows, we update `invertColors` to take a pointer. Then, we pass the pointer of our array to the function.

Listing 2.5 Passing arrays to functions with pointers (arrays2/array.go)

```
package main

const resolution8K = 7_680 * 4_320 * 3

func main() {
    selfie := [resolution8K]byte{ /* pixel data */ }
    invertColors(&selfie)                       #A
}

func invertColors(colors *[resolution8K]byte) {        #B
    for i := range colors {
        colors[i] = 255 - colors[i]            #C
    }
}
```

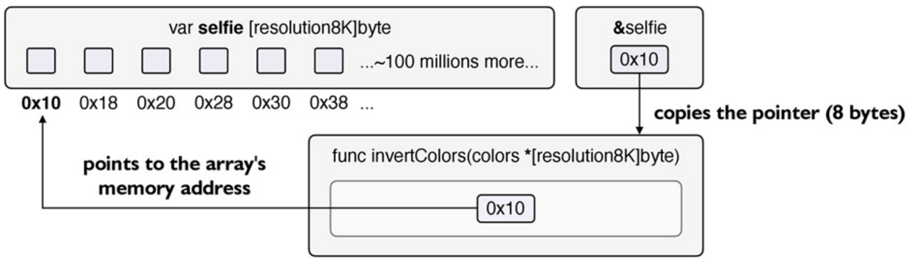
Note

Calling this function only copies the array's pointer, not the array's entire elements.

As Figure 2.8 shows, passing the array's pointer to the function only copies 8 bytes (pointers are always 8 bytes on a 64-bit machine) instead of the entire array. Moreover, the function can mutate the original array's elements via the pointer without copying.

This approach is much more efficient than passing and copying a large array.

Figure 2.8 Passing the array's pointer to the function only copies 8 bytes.


var selfie [resolution8K]byte
...~100 millions more...
0x10 0x18 0x20 0x28 0x30 0x38 ...
&selfie
0x10
copies the pointer (8 bytes)
func invertColors(colors *[resolution8K]byte)
0x10
points to the array's memory address

This pointer points to the array's memory address. We could also point to the individual array elements with separate pointers (e.g., `&selfie[1]` points to the second element), but we don't need it here. However, we rarely use arrays; slices are more useful in practice.

Exercises

Note

Imagine what the data looks like for these exercises; it doesn't have to be real-world data.

- Write a function to calculate total sign-ups from a daily user counts array.
- Pass a pointer to a function to update network response speeds array.

Wrap up

- Arrays (i.e., `[Length]ElementType`) are fixed-length collections of elements.
- Array's length and element type fuses together to determine its type.
- `len()` returns the array's length.
- Passing arrays around copies the elements in the array to a new array.
- Passing large arrays around by pointers *can* be more efficient.

Dive deep: Arrays

Since their lengths and element types are fixed, arrays always occupy the same memory space. We should recompile our programs if we want an array to hold more (or less) elements. Arrays would waste memory space if we have a larger array than we need.

The bright side of arrays is that their elements are consecutively arranged in memory. This is useful for the CPU caches because it's more efficient for a CPU to access (and re-access) a set of elements that can fit into one of the CPU caches. See the wikipedia article for more details about CPU caches: https://en.wikipedia.org/wiki/CPU_cache.

2.3.2 Slices

A slice is a dynamically-sized, flexible view into the elements of an array.

Unlike fixed-length arrays, they are declared without a length:

```
var hits []int

len(hits) // 0
```

A slice's zero value is nil since it has an implicit pointer (we'll see what that means soon). So, this is a `nil` integer slice with a zero length. Let's initialize a slice with elements:

```
rgb := []byte{214, 65, 79}    #A

rgb[0]          // 214        #B
rgb[len(rgb)-1] // 79        #B
```

Slice usage is similar to array usage. However, there is a crucial difference. We can effectively use slices only when we grasp how they internally work. Let's dive in.

Tip

The lengths of both `nil` (e.g., `var hits []int`) and empty slices (e.g., `hits := []int{}`) are zero. However, the latter would return `false` if we check if it's `nil`. To avoid surprising results, always use `len` (e.g., `if len(hits) == 0`) to check if a slice is empty.

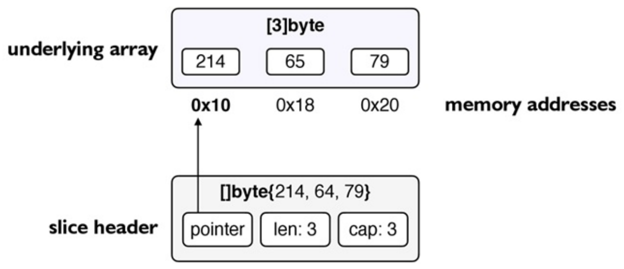
Bound checks prevent memory overflows

Go ensures memory safety with automatic bounds checks at the assembly level, preventing memory overflow attacks by crashing the program when access beyond a slice's capacity is attempted. The compiler omits these checks for efficiency when it verifies our code is safe. Discover more in this video: https://www.youtube.com/watch?v=5toTS6kSWHA

What's a slice?

Slices do not directly store their elements. As Figure 2.9 shows, a slice represents (or describes) a segment of an underlying array that stores the elements in memory.

Figure 2.9 A slice and its underlying array. Since the slice's length is three and points to the array's first element, it sees the array's entire elements.



Behind the scenes, a slice is a "slice header" managed by Go Runtime that includes:

- *Pointer*—Pointer to the slice's underlying array.
- *Len*—Length defining the accessible array's elements through the slice.
- *Cap*—Capacity indicates how long a slice can be extended (more on this soon).

A slice can see the elements of the underlying array based on the slice's pointer and the slice's length (pointer + length). In our case, the `rgb` slice can see the entire array.

Passing slices around is cheap

Separating representation (slice) from data (underlying array) makes slices cheap to copy (a slice is only 24 bytes long on a 64-bit machine). Hence, passing a slice to a function only copies the slice header instead of the underlying array (even if the array has a million items).

Listing 2.6 Passing a slice to a function (slices1/slice.go)

```
package main

func main() {
    selfie := []byte{ /* ~100 MB pixel data */ }
    invertColors(selfie)            #A
}

func invertColors(colors []byte) {        #B
    for i := range colors {
        colors[i] = 255 - colors[i]    #C
    }
}
```

Passing this slice to the function is cheap and only copies 24 bytes of memory (the slice header) instead of the slice's underlying array (which occupies ~100 MB in memory).

Note

A function can mutate the slice's underlying array via the given slice's implicit pointer.

Slice expressions and capacity

Now that we've seen that slices are descriptors for an underlying array, let's delve into how to shrink and extend slices. One way to do that is to slice a slice using *slice expressions*:

```
slice[lowestIndex : highestIndex : capacity]    #A
```

Say we use a slice to hold website request counts. As the next listing shows, using *slice expressions*, we can slice a slice to easily pick the first and last two request counts.

Listing 2.7 Slicing a slice (slices2/slice.go)

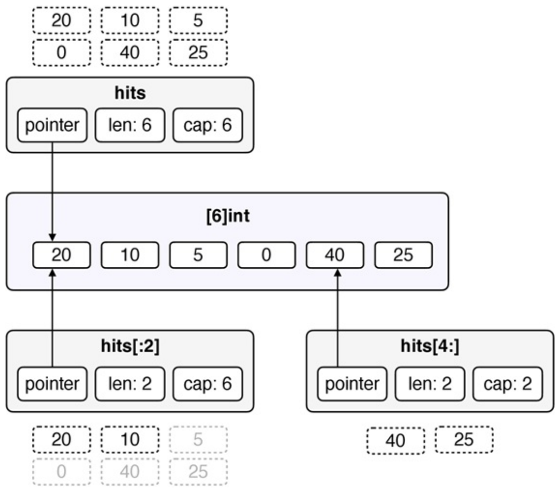
```
package main

import "fmt"

func main() {
    hits := []int{20, 10, 5, 0, 40, 25}

    firstTwo := hits[:2]            #A
    lastTwo  := hits[len(hits)-2:]        #B

    fmt.Println(firstTwo) // prints [20,10]
    fmt.Println(lastTwo)  // prints [40,25]
}
```

We've created two new slices (`firstTwo` and `lastTwo`) after slicing the original `hits` slice. While doing so, we omitted some of the optional indexes from slice expressions:

- `hits[:]` returns a new slice that is equal to `hits[0:len(hits)]`
- `hits[:2]` returns a new slice that is equal to `hits[0:2]`
- `hits[4:]` returns a new slice that is equal to `hits[4:len(hits)]`

Figure 2.10 shows how the slice headers and their underlying array look behind the scenes.

Figure 2.10 Slices shown describe a segment of the same underlying array (`[6]int` in the middle) that stores the elements. Slices do not store the elements.



The slices we declare are a segment of a different portion of the same underlying array. The dashed boxes represent how slices see their underlying array. The faded boxes are the elements we can access through the slice if we extend the slice. We cannot extend these slices except the slice in the bottom left because their capacities are full (their lengths are equal to their capacities). Let's extend the one we can extent (the bottom left one):

```
firstTwo := hits[:2]
extended := firstTwo[:6]
// or extended := firstTwo[:cap(firstTwo)]
```

The extended slice looks like this:

```
[20, 10, 5, 0, 40, 25], len:6, cap:6
```

This new slice's length and capacity are six, and it sees the entire underlying array elements. We can no longer extend the new slice beyond this point because its capacity is full.

Before exploring how to create a new slice with a larger capacity, let's look at a spooky action at a distance. Since these slices share the same underlying array, all of them will see the updates if we mutate an element via one of these slices. Consider this:

```
firstTwo[0] = 100
fmt.Println(hits[0], extended[0]) // 100 100
```

We have the option to prevent this by disconnecting the underlying arrays. More on this later.

Appending

Let's get back to the topic of the capacity of a slice. The idiomatic way to increase a slice's capacity is to use the built-in `append` function.

```
append(s []T, e ...T) []T    #A
```

Note

Appending returns a new slice with a larger capacity without mutating the input slice.

This function takes a slice and an arbitrary number of values and returns a new slice with a larger capacity. It always allocates a new underlying array if the slice's capacity is full. Otherwise, `append` will add the new elements to the same underlying array. Consider this.

Listing 2.8 Appending to a slice (slices3/slice.go)

```
...
func fetchHits() []int {
    var hits []int            #A

    for i := range 4 {            #B
        hits = append(hits, i + 1)        #C
    }

    return hits
}
```

Calling this function will return the following slice:

```
[1 2 3 4] len:4 cap:4
```

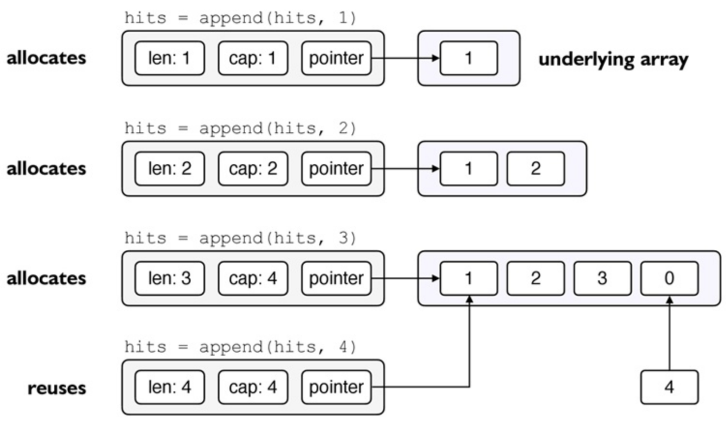
This example simulates fetching incoming website request counts, appending them to a slice, and returning the result slice. We start with a `nil` slice (zero length and capacity). Since appending returns a new slice, we keep overwriting the same slice variable during the loop.

Warning

The changes won't be visible outside if we don't return the slice from the function. This is different from mutating the elements. Here, we're mutating the slice variable itself.

The crucial point is that if the slice doesn't have enough capacity, `append` allocates a new array. Otherwise, it reuses the existing one. Figure 2.11 shows how `append` works.

Figure 2.11 Append allocates an array or reuses it depending on capacity.



Due to lacking capacity, `append` allocates new arrays for the first three operations and copies the elements from the previous arrays. The fourth `append` reuses the same array and adds the last element to the same array since there is enough capacity in the third slice.

Note

The `append` function doubles the array length until 256 elements, then grows arrays by ~25% (eventually converges into 25%). We'll see how to reduce allocations soon. See the link if you're interested in the internals: https://go-review.googlesource.com/c/go/+/347917

When appending, we should overwrite the existing slice with the returned one to avoid surprising behavior. This is because `append` may allocate a new array and return a slice with a pointer to the new array. If we don't update the original slice with the returned one, it might point to the old array, not the updated one, causing surprising results and bugs.

Tip

See the link to learn about creative usages of appending: https://go.dev/wiki/SliceTricks

Making slices

Instead of letting `append` to frequently allocate arrays, we can preallocate an array once:

```
make([]T, length, capacity) []T
```

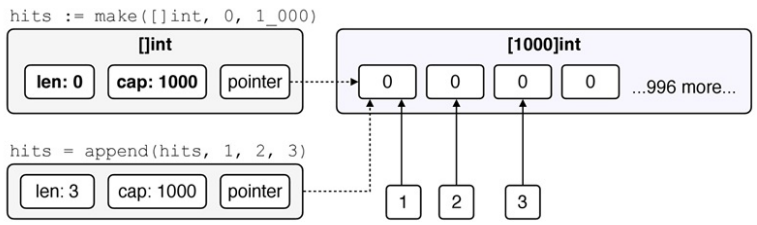
This built-in function creates a slice with a new array with the specified length and capacity. In the next listing, we create an `int` slice with 1000 elements. Then, we append multiple elements to the slice twice. None of these appends will allocate a new underlying array.

Listing 2.9 Pre-allocating a slice (slices5/slice.go)

```
package main

import "fmt"

func main() {
    hits := fetchHits()
    fmt.Println(hits)        #A
}

func fetchHits() []int {
    hits := make([]int, 0, 1_000)    #B
    hits = append(hits, 1, 2, 3)    #C
    hits = append(hits, 4, 5, 6)    #C
    // ...continue appending more elements...
    return hits
}
```

As Figure 2.12 shows, we allocate an array once and get an `int` slice with a pointer to that array. The slice's capacity is one thousand, so appending won't allocate. But the returned slice's length is zero. This allows us to append the elements to the beginning of the array.

Figure 2.12 Appending won't allocate a new array if a slice has sufficient capacity.



If we didn't pass zero to the second input (specifies the length), `make` would return a slice both with a length and capacity of one thousand. Appending to the slice would allocate a new underlying array and append the element to the end of the new array as the 1001st element:

```
hits := make([]int, 1_000)
hits = append(hits, 42)
```

The slice's array would contain `[...1000 elements..., 42]`. We should omit the third input (specifies the slice's capacity) if we only want to append elements to the array's end. Omitting the length is useful when we want to directly write into the slice using indexes:

```
hits[0], hits[1] = 1, 2
```

Remember

The `append` function appends the elements to the end of a slice.

Slicing arrays

We can slice an existing array to get a slice:

```
rgb   := [3]byte{41, 190, 176}
bytes := rgb[:2] // []byte{41, 190}
```

Note

Mutating the slice will mutate the array because the array is the slice's underlying array. For instance, running `bytes[0] = 214` will make the `rgb`'s first element to become 214.

Slicing is useful if we want to pass the array as a slice to a function that expects a slice.

For instance, say we want to detect if an image is a PNG image. Our task is straightforward because the first eight bytes of every PNG image start with the same set of bytes called the "PNG header." This `isPNG` function in the next listing detects if `src` has a PNG format.

Listing 2.10 Detecting the PNG format (slices6/slice.go)

```
package main

import (
    "fmt"
    "slices"
)

func main() {
    buffer := []byte{
        137, 80, 78, 71, 13, 10, 26, 10, 65,
        /* ..more.. */
    }
    fmt.Println(isPNG(buffer)) // prints true
}

var pngHeader = [8]byte{                    #A
    137, 80, 78, 71, 13, 10, 26, 10,
}

func isPNG(src []byte) bool {
    if len(src) < len(pngHeader) {                #B
        return false
    }
    return slices.Equal(src[:len(pngHeader)], pngHeader[:])    #C
}
```

The last line returns `true` if two slices are equal. For the first input, we slice the bytes to get the first eight bytes (`src[:len(pngHeader)]`). For the last input, we make a slice out of the array (`pngHeader[:]`) that describes the entire array elements (`len` and `cap` is 8).

Since slicing beyond the slice's capacity will panic, before slicing `src`, we check the lengths of the input slice and the PNG header array to avoid panics (`if len(src) < len(pngHeader)`).

We can also slice strings as they are immutable byte slices. This also means we can convert a byte slice to a string (or vice versa). For instance, `[]byte("\x89PNG\r\n\x1a\n")` returns an equal byte slice (element-wise) to the `pngHeader` slice we declare above.

Exercise

Detect the PNG format of a byte slice using the following string variable instead of an array: `var pngHeader = "\x89PNG\r\n\x1a\n"` (`\x` encodes a hexadecimal number to a byte).

Copying slices

Earlier, we discovered that slices that share the same underlying array will see updates from each other when an element mutates via one of the slices. For example:

```
server := []byte("authorization server v:1")    #A
version := server[len(server)-3:]            #B
version[len(version)-1] = byte('2')        #C
```

We convert a `string` into `[]byte`. Go creates an underlying byte array to store the bytes in the string and a slice with a pointer that points to that array. Then, we slice the version part. Lastly, we mutate the last element of the version slice to two. Now, see this:

```
fmt.Println(string(server))
```

This will print "authorization server v:2". Although we update the version slice's last element, the server slice also sees the updated version. We can copy the slices if we don't want this.

There is a built-in `copy` function that copies slices and returns the number of elements copied (we should be careful that `copy` only copies the minimum lengths of the slices).

```
func copy(destination []Type, source []Type) int
```

Let's consider this:

```
server := []byte("authorization server v:1")
version := make([]byte, len(server))    #A
copy(version, server)            #B
```

Note

We omit the make's capacity argument (the third one) to get a slice with the `server` slice's length. If the `version` slice's length were `nil`, `copy` wouldn't copy anything.

We create a byte slice with the same length as the `server` slice. Then, we copy the entire bytes in the `server` slice to the `version` slice. Let's look at the results:

```
version[len(version)-1] = byte('2')
fmt.Println(string(server))  // prints authorization server v:1
fmt.Println(string(version)) // prints authorization server v:2
Each slice now uses a separate underlying array (make returns a s
```

Exercises

- Create a slice for daily login counts, append, print, and then slice for days. Each index in the slice should represent the day number in a year.
- Loop over a string slice of server endpoints, printing each.
- Make a slice for server response times, populate and print.
- Append to a byte slice, reslice, and note capacity changes.
- Remove bytes from a `byte` slice and reduce the slice's capacity by one.
- Convert a string to a byte slice, modify, and revert to string.
- Sum database query counts from an integer slice with a summing function.
- Append one slice of response times to another, return the result.
- Remove duplicate numbers from an integer slice.

Tip

You can find helper functions for common slice operations (e.g., `Max`, `Reverse`, etc.) in the `slices` package at the link: https://pkg.go.dev/slices.

Wrap up

- A slice (i.e., `[]ElementType`) is an underlying array's representation.
- `len()` returns the slice's length, whereas `cap()` returns the slice's capacity.
- A slice header describes the underlying array's segment.
- Passing slices around only copies the slice header (not the underlying array).

- Mutating the slice elements mutates the underlying array elements.
- Slice expressions (`[low:high]`) can shrink and expand a slice (limited by the slice's capacity). Full slice expressions (`[low:high:max]`) can limit the slice's capacity.
- Appending appends elements to the slice's end and returns a new slice. This operation allocates a new underlying array if the slice lacks enough capacity.
- The `make` function returns a slice with a pre-allocated underlying array.
- Like slices, arrays and strings can also be sliced using slice expressions.
- The `copy` function copies slices and returns the number of elements copied.

2.3.3 Maps

A map is a collection of key-value pairs. Keys can be any *comparable type*, and values can be of any type. Map operations such as adding and looking up typically run in constant time.

Note

The equal operator (==) can compare any comparable type. For instance, we can compare strings and numbers, but we cannot compare slices, maps, and function values.

For instance, we can use a map to track web server memory usage:

```
var usage map[string]float64
```

This map's type is a `map[string]float64` (its key type is `string`, and its element type is `float64`). But it's a `nil` map (uninitialized), and adding elements to this map will panic:

```
usage["gateway"] = 42.5
// panic: assignment to entry in nil map
```

We can initialize a map with either of the following (the first is simpler and more common):

```
usage := map[string]float64{} // or:
```

```
usage := make(map[string]float64)
```

Adding elements to this map no longer panics. See the next listing for an extended example.

Listing 2.11 Using a map (maps/map.go)

```
package main

import "fmt"

func main() {
    usage := map[string]float64{            #A
        "gateway": 75.,
        "auth": 50.5,
    }

    usage["health"] = 99.0            #B
    fmt.Println(usage["health"])    // prints 99

    fmt.Println(len(usage))    // prints 3

    delete(usage, "health")            #C

    v, ok := usage["health"]
    fmt.Println(v, ok)        // prints 0 false

    clear(usage)
    fmt.Println(len(usage))       // prints 0
}
```

Note

A map returns the zero value of their element type when a key doesn't exist.

Here are the map operations:

- Use square brackets to set and retrieve elements in a map.
- Maps return the zero value of their element type if a key doesn't exist.
- Use the `delete` function to remove an element.
- Use `len` to get the number of elements in the map.
- Use `clear` to remove the map elements (it also works for slices but instead of removing the elements, it sets a slice's elements to their zero

value).
- Map iteration order is undefined (not random).

For instance, the following might print a different output on each run:

```
for key, value := range usage {
    fmt.Print(key, ":", value, " ")
}
// health:99 gateway:75
// gateway:75 health:99
```

Printing a map is convenient for getting sorted output for debugging purposes.

```
fmt.Println(usage) // map[gateway:75 health:99]
fmt.Println(usage) // map[gateway:75 health:99]
```

A map is a pointer to an internal data structure managed by Go Runtime. Functions can mutate maps (adding or removing elements), and the changes will be visible outside.

Exercise

Pass a map to a function, mutate it, and print the mutated map from the caller.

Note

For internals of maps, visit: https://www.youtube.com/watch?v=Tl7mi9QmLns

Tip

You can find helper functions for common map operations (e.g., `Equal` to check if two maps are equal) in the `maps` package at the link: https://pkg.go.dev/maps.

Exercises

- Use `make` to create a map for user IDs and names, set and retrieve

values.
- Check a non-existent user ID in the map, noting the zero value return.
- Remove a user ID and name entry with `delete`, then use `len` to show map size.
- Iterate over the user ID map, printing each key-value pair, noting the order.
- Increment login counts in a map with user IDs and login counts.

2.4 Object-oriented programming

Object-oriented programming in Go is *different* than usual.

Go ditches classes and inheritance. We're free to attach methods to any concrete type as long as both are in the same package (e.g., we cannot attach methods to `time.Duration` unless we redefine it in our package). Types can implicitly implement interfaces.

2.4.1 Structs

Imagine our company's reliability team is requesting a monitoring program to track the servers and receive notifications about server response times, particularly for the slow ones. For instance, to represent a server in the monitoring program, we can use a struct type.

The `server` struct type with a `url` and `responseTime` fields represents a server. The `url` field is a `string`, while the `responseTime` field is Go's `time` package's `Duration` (a type that stores a time duration as an integer). The next listing declares the `server` struct type.

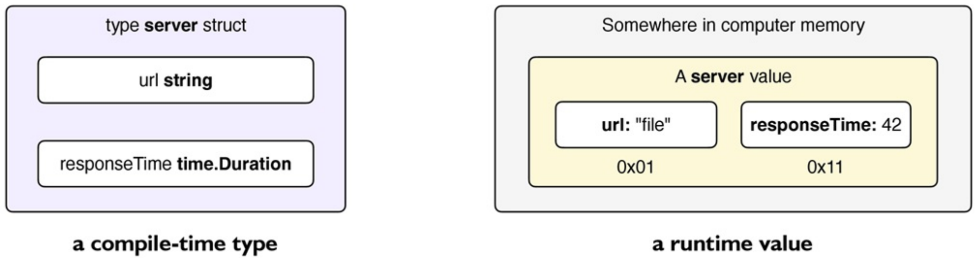
Listing 2.12 Declaring a struct (server/server.go)

```
type server struct {        #A

    url          string        #B
    responseTime time.Duration    #B
}
```

Structs are similar to classes but without inheritance. They're raw data types that can store various types of values in their fields. Just as we use classes to instantiate objects in other object-oriented programming languages, we can create values from struct types in Go.

Figure 2.13 shows an example. We create a `server` value using the `server` struct type.

Figure 2.13 Struct fields are arranged consecutively in memory and, like variables, have memory addresses. A struct's address is the address of its first field (if any).



As the next listing shows, we can now create a `server` value and then assign that value to a variable (`fileServer`). The `%s` verb in `Printf` prints a string, while `%d` prints an integer.

Listing 2.13 Using a struct value (server/main.go)

```
func main() {
    fileServer := server{url: "file"}        #A

    fmt.Printf(                         #B
        "url: %s response time: %d\n",        #B
        fileServer.url, fileServer.responseTime,    #B
    )                           #B
}
```

We've declared a variable named `fileServer` with the `server` struct type. This variable's `url` field is equal to "file," and the `responseTime` field is zero. When we omit some or all fields of a struct while creating it, the compiler sets the omitted fields to their zero values.

Note

Go doesn't have built-in constructors. Instead, we can use a literal or a function to initialize a value. We'll cover constructors later in the book.

Underlying types

Named types must be declared from another type as its underlying type. For instance, we declare the `server` type from an anonymous struct type (the part highlighted in bold in Listing 2.12). The `server` type is based on an anonymous struct, but we can also declare types from named types. For instance, let's look at the following `Duration` type:

```
// package time (https://pkg.go.dev/time#Duration)
type Duration int64
```

The underlying type of this type is `int64`, a 64-bit integer.

A type's underlying type determines its memory layout and permissible operations: we can do arithmetic on `Duration` values because `Duration` is an `int64` type with a new name. Still, named types are distinct from their underlying types. To use them together, we must *explicitly convert* between `int64` and `Duration` values. We'll see this in action soon.

Exercises

- Initialize a `server` with a URL and response time, then print.
- Create a slice of `servers`, print each `server`'s URL and response time.
- Populate a map with URLs and `servers`, retrieve and print an element.
- Add or replace a `server` in the map and demonstrate retrieving the updated value.
- Write a function that updates a `server`'s response time (use a pointer).
- Write a function that calculates average response time from a slice of `servers`.
- Write a function that sorts a slice of `servers` by response time.
- Write a function that deletes a `server` from the map by URL.
- Write a function to divide response times by two to simulate improved times.

2.4.2 Methods and receivers

Methods are ordinary functions attached to a type with a *receiver*—a special input parameter before a function's name. Think of receivers as `this` or `self` in other object-oriented programming languages. There are two kinds of receivers: Pointers and value.

- Methods with pointer receivers can mutate the value pointed by their receiver.
- Methods with value receivers can only mutate their local receiver.

Methods with pointer receivers are more similar to the methods in other object-oriented programming languages. Let's look at an example to understand methods and receivers.

Pointer receivers

Imagine we want to check a server's response time and determine whether it's responding slowly. Once we check the response time, we'll set the `server`'s `responseTime` field.

Since we'll mutate this field, we'll add our methods with a pointer receiver instead of a value receiver. As the next listing shows, we have two methods: `check` and `slow`.

Listing 2.14 Attaching methods to the server type (server/server.go)

```
func (s *server) check() {                #A

    s.responseTime = 3 * time.Second            #B
}

func (s *server) slow() bool {                #A
   return s.responseTime > 2 * time.Second            #C
}
```

The `s` inside the parentheses is called the receiver, and `*server` is the receiver's type. The receiver `s` is like an input parameter and becomes a local variable within the method. For instance, within the `check` method, this variable's name is `s` and type is `*server`.

```
func (s  *server) check
      ^  ^^^^^^^
      |   |
receiver  receiver's type
```

The `check` method receives and uses `s` to update the `responseTime` field. Although the `slow` method doesn't mutate its receiver, we also attached `slow` as a pointer receiver for consistency with `check`. Consistent usage of receiver types wards off bugs and confusion.

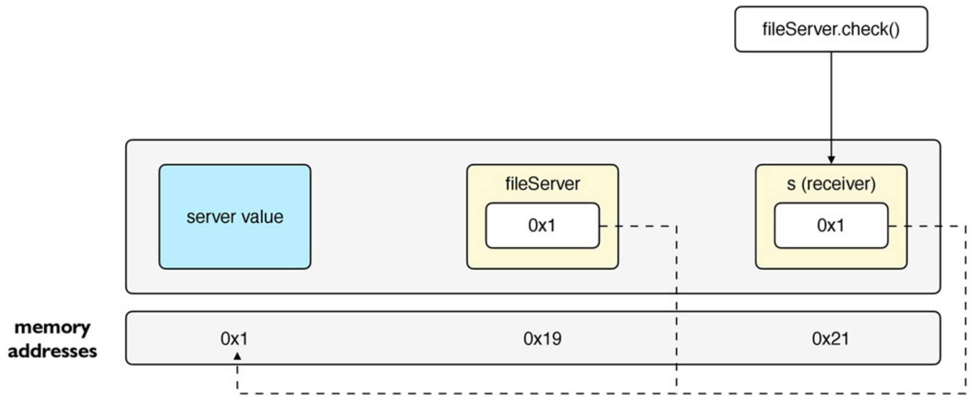
We can now call the `server`'s methods. See the next listing for an example where we declare a pointer to a `server` value. Then, we call `check` to update the `responseTime` field.

Listing 2.15 Using a struct value (server/server.go)

```
func main() {
    fileServer := &server{url: "file"}        #A
    fileServer.check()                #B

    fmt.Printf("is slow? %t\n", fileServer.slow())    #C
}
```

Initially, the responseTime is "0s." Once we call `check` on `fileServer`, it receives and copies `fileServer` to a local variable just like an input parameter. Since this local variable is a pointer (`*server`), the `check` method can mutate the original `server` value's `responseTime` field. So, the `slow` method returns `true` instead of `false`. See Figure 2.14 for an illustration.

Figure 2.14 When we call `check` on `fileServer`, the compiler copies the memory address in `fileServer` into the local variable named `s` within the `check` method.



Like function parameters, receivers are passed-by-value. When we call `check`, the compiler passes a copy of the receiver (`s` in Listing 2.14) to the `check` method. Since the copied pointer points to the original `server` value, the method can update the `responseTime` field.

Note

Since we attach the `check` method with the pointer receiver, only the `*server` type has the method, but the `server` type doesn't have it. This is called the method set of a type. While `server`'s current method set is empty, `*server`'s method set has a `check` method. E.g., method sets determine which methods we can use to call on an instance of a type.

Structs are value types

Unlike objects in some other languages, *structs are value types* copied when passed around. So, if we had attached the `check` method with a value receiver, it wouldn't mutate the field:

```
func (s server) check() {            #A

    s.responseTime = 3 * time.Second    #B
}

// ...

fileServer.check()
fmt.Println(fileServer.responseTime) // prints 0s
```

To wrap up, receivers are similar to function input parameters. They can mutate the original values when they are pointers. Let's switch focus to value receivers for a complete grasp.

Value receivers

We can use value receivers if we want to work with immutable values.

Imagine we want to track server CPU usage. The next listing declares the `usage` type.

Note

Methods can be added to any concrete type in the same package, not just structs.

Listing 2.16 Methods with value receivers (server/usage.go)

```
type usage float64                    #A


func (u usage) high() bool     { return u >= 0.95 }        #B

func (u usage) set(to float64) { u = usage(to)    }        #C

func main() {
    cpu := usage(0.99)                    #D
    fmt.Println("high usage?", cpu.high()) // ..true

    cpu.set(0.7)                           // cpu is still 0.99
    fmt.Println("high usage?", cpu.high()) // ..true
}
```

The `high` method detects high usage, but this initial version of the `set` method incorrectly assumes to mutate the original `usage` value—it can't, as *methods with value receivers can't mutate the original value* they're called on. They can only mutate the local copy of the receiver they receive, which is not visible to the outside.

If we want to mutate the `usage` value, we should return a new one. Let's rewrite `set`.

```
func (u usage) set(to float64) usage { return usage(to) }
```

Now, to apply this new value returned by the method, we must explicitly reassign it.

```
cpu = cpu.set(0.7)    #A

fmt.Println("high usage?", cpu.high()) // ..false
```

Note

Go's `Time` type's `Add` method does the same:
https://pkg.go.dev/time#Time.Add.

To wrap up, receivers are copied when methods are called.

Which receiver type to choose?

We might choose a receiver type based on *how our values will be used*. Pointer receivers are great for rarely created, often shared, and mutable values. Value receivers, on the other hand, are perfect for frequently created, primitive, and immutable values.

We should consider optimizing for performance only if it's a concern. Value receivers are more CPU cache-friendly, while pointer receivers, although efficient in passing only a memory address, can burden the garbage collector with tracking all these references. Still, we should *avoid prematurely optimizing before measuring*, like picking a value receiver just to reduce garbage collection pressure or a pointer receiver for efficient data passage.

Exercises

- Write a pointer receiver method to reset a `server`'s response time.
- Use a value receiver to check if the CPU `usage` is within the threshold.
- Implement a method on `usage` that scales the value, returning a new usage.
- Define a new `cache` type representing a key-value store with a map. Attach a pointer receiver method that adds or updates entries in the cache.
- Create a `metric` type that holds server performance data. Attach a value receiver method that returns a summary report of the metrics.

2.4.3 Implicit interfaces

Interfaces in Go are similar to other languages. The power of interfaces is in abstracting *shared behavior*. They enable us to write general-purpose code that can operate on a set of different types, as long as those types have *all* the methods defined by the interface.

Starting with the concrete types

Say we want to notify our company's reliability team over Slack and SMS when servers in production slow down. We can generalize the sending of notifications using an interface.

However, it's rare in Go to declare interfaces *upfront* because types *implicitly* satisfy them.

So, we'll first implement our concrete types without considering whether we'll satisfy an interface at this stage. In the next listing, we declare two types for sending notifications. Each has a `notify` method that takes a string. They simulate sending a notification.

Listing 2.17 Implementing the notifiers (server/notify.go)

```
type slackNotifier struct {            #A

    apiKey  string
    channel string
    // ..other fields
}

func (s *slackNotifier) notify(msg string) {    #B
    fmt.Println("slack:", msg)
}

func (s *slackNotifier) disconnect() {
    fmt.Println("slack: disconnecting")
}

type smsNotifier struct {
    gatewayIP string
    // ..other fields
}

func (s *smsNotifier) notify(msg string) {        #B
    fmt.Println("sms:", msg)
}
```

The `slackNotifier` is a concrete type that has `notify` and `disconnect` methods. The `smsNotifier`, on the other hand, has only a single `notify` method. Now that we have two notification ways, we can write a function to send notifications if a server is slow.

```
func notifySlack(s *server, slack *slackNotifier) {

    if !s.slow() {
        return
    }
    slack.notify(...)
}

func notifySMS(s *server, sms *smsNotifier) {
    if !s.slow() {
        return
    }
    sms.notify(...)
}
```

Because Go is a statically typed language, we can't interchangeably use a `slackNotifier` in place of an `smsNotifier`. So, we've written functions that take separate notification types.

Discovering an interface

However, this approach won't scale as the team plans for new notification ways. Instead, we can write a single function if we can find the shared behavior. Since both notifiers have a `notify` method, we can describe this with an interface. As the next listing shows, we're declaring the `notifier` interface with a `notify` method but without an implementation.

Listing 2.18 Implementing the notifier interface (server/notify.go)

```
type notifier interface {

    notify(message string)    #A
}
```

This interface type lays out what can be done but not how it's specifically done—that's the job for *concrete types*, like a Slack or SMS Notifier. Our concrete notifiers implicitly satisfy the `notifier` interface because each has a `notify` method with the same signature.

Note

We use is-a when referring to a type that implements—satisfies—an interface. For instance, a `slackNotifier` type is a notifier because it satisfies the `notifier` interface.

As in the next listing, we can now write a single function to send a notification rather than two. We can pass any `notifier` to this function. As long as a type implements *all the methods* defined by an interface, it can be used wherever that interface is expected.

Listing 2.19 Implementing the notify function (server/notify.go)

```
func notify(s *server, n notifier) {    #A

    if !s.slow() {
        return
    }
    msg := fmt.Sprintf(
        "%s server is slow: %s",        #B
        s.url, s.responseTime,
    )
    n.notify(msg)                 #C
}
```

Note

We declare interfaces only when a consumer wants to switch between different types. In our case, the `notify` function is the consumer of the concrete notifier types.

This function focuses solely on the notification behavior rather than the specific concrete types. It can work with any notifier and doesn't mind how the notification is delivered.

Wiring up

Now that we have everything in place, let's wire everything up in the next listing.

Listing 2.20 Gluing everything together (server/notify.go)

```
func main() {
    authServer := &server {
        url: "auth",
        responseTime: time.Minute
    }
    slack := &slackNotifier { /* Slack specific configuration */
    sms   := &smsNotifier   { /* SMS specific configuration   */

    notify(authServer, slack)
    notify(authServer, sms)
}
```

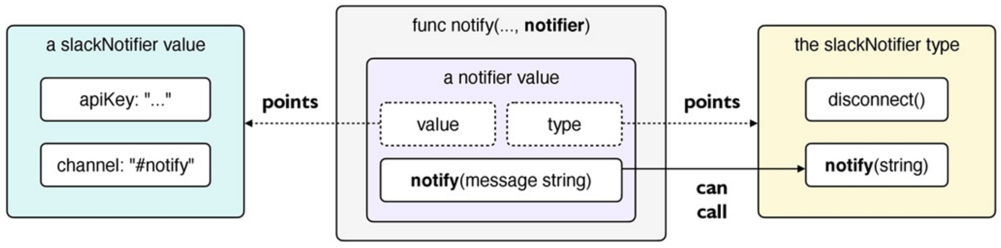
Once we run this program, it'll print the following:

```
$ go run .
slack: auth server is slow: 1m0s
sms: auth server is slow: 1m0s
```

Thanks to the interface, we can now use the same function while sending notifications. The interface approach boosts the modularity of our program and reduces duplication.

As Figure 2.15 shows, from the `notify` function's perspective, the concrete notifier types (`slackNotifier` and `smsNotifier`) are interface values with a single `notify` method (even though `slackNotifier` has an additional `disconnect` method).

Figure 2.15 A Slack Notifier value appears as a `notifier` value within the function. This is a simplified view as an interface value points to a copy of the original value. Keep in mind that a notifier interface value can point to any notifier, not just Slack.



Its internal value and type fields are pointers and set in runtime depending on what we pass. This shape-shifting capability makes interfaces powerful. Visit

the link to learn more about interface values:
https://research.swtch.com/interfaces.

Harnessing the power of interfaces

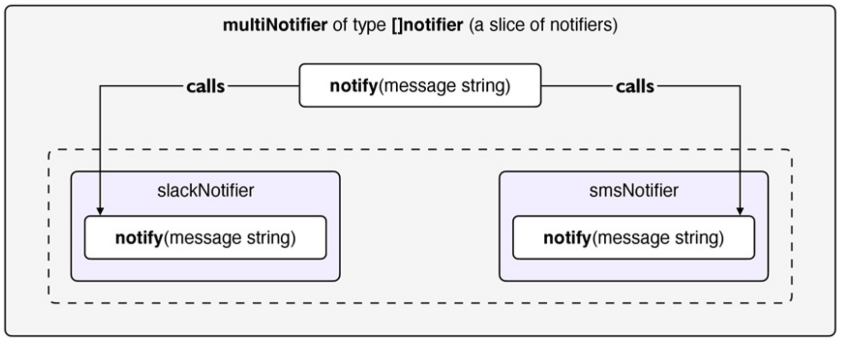
Calling the `notify` function each time can be a hassle and error-prone if we have many notifiers. Instead, we can group them with a slice and call `notify` in a loop.

```
// say this code is at the end of main in Listing 2.20
for _, n := range []notifier{slack, sms} {
    notify(authServer, n)
}
```

This loop skips the index variable with a *blank identifier* (underscore) and declares only a `notifier` variable (`n`). Each iteration *recreates* `n` and *copies* the next `notifier` to `n`.

Instead of using a loop the previous way, we can do better by harnessing the versatility of interfaces. Figure 2.16 shows a new `notifier` implementation. `multiNotifier` forwards a call to its `notify` method to the other notifiers it has, making them send notifications.

Figure 2.16 `multiNotifier` forwards a `notify` call to the notifiers it stores.



Let's put this in action. In the next listing, we declare `multiNotifier` based on `[]notifier`. Then, we add the `notify` method to satisfy the `notifier` interface.

Listing 2.21 Implementing multiNotifier (server/notify.go)

```
type multiNotifier []notifier            #A

func (m multiNotifier) notify(msg string) {    #B
    for _, n := range m {                #C
        n.notify(msg)                #C
    }
}
```

Recall that a type acts like its underlying type.

Since a `multiNotifier` is a slice, we can loop over it and call the `notify` methods. Because `multiNotifier` has a `notify` method, we can pass `multiNotifier` as any other `notifier` to the `notify` function. See the next listing.

Listing 2.22 Running with multiNotifier (server/notify.go)

```
func main() {
    authServer := &server{url: "auth", responseTime: time.Minute
    slack      := &slackNotifier{ /* ... */ }
    sms        := &smsNotifier  { /* ... */ }

    notify(authServer, multiNotifier{slack, sms})    #A
}
```

Thanks to `multiNotifier`, we can alert the team with many notifiers without changing the existing code. Although we pass a single `notifier`, `multiNotifier` calls others for us:

```
slack: auth server is slow: 1m0s
sms: auth server is slow: 1m0s
```

Interfaces make our code testable and flexible. For instance, we might pass a *fake notifier* to test the `notify` function without sending notifications. Or, if the team wants other notification ways, we're ready to adopt them as our function focuses solely on delivering notifications without being tied to the specifics. Despite the benefits of interfaces, in Chapter 11, we'll see when to avoid interfaces, as they can complicate our code and make navigation difficult.

Exercises

- Declare two types that simulate logging to a file and console. Add methods to both types; once called, they should print a log line. Add more methods if necessary.
- Discover and declare an interface that defines the shared behavior.
- Write a function that can be called with any of the types that you declared.
- Declare a type that can log output with multiple logger types.

2.4.4 Wrap up

This section served as a primer on object-oriented programming in Go, giving us the building blocks to write flexible programs. We'll dive into this more later. Here's a wrap-up.

- Structs bundle types into a single type where each field is a variable at runtime.
- Methods are functions tied to a concrete type. Pointer receiver methods can alter the original values pointed by their receiver, while value receiver methods cannot.
- Interfaces abstract common behavior and are satisfied implicitly by concrete types.

A tour of generics, type parameters, and type constraints

Generics in Go are not much different from other languages. They let us write code that works uniformly across various types, reducing the need for repetitive code. Go uses square brackets `[T type]` instead of angled `<T type>` to define type parameters for easy parsing.

Say we want to calculate the average of varied types, such as response times (the `Duration` type) and CPU usage (the `usage` type). We can declare these two functions to do so:

```
func avgTimes (nums []time.Duration) time.Duration { ... }

func avgUsages(nums []usage) usage                 { ... }
```

Instead, generics empower us to write a single function, `avg`:

```
func avg[T number](nums []T) T {    #A

    var t T             #B
    for i := range nums {        #C
        t += nums[i]
    }
    return t / T(len(nums))    #D
}
```

Our generic function takes a slice of `T` and returns a single `T` value. It has a generic type parameter: `[T number]`, where `T` can be of any type that satisfies the `number` interface. We can use interfaces not only to define shared behavior but also to constraint a set of types:

```
type number interface {        #A

    ~int | ~int64 | ~float64    #B
}
```

The `int`, `int64`, `float64` types, or any types based on these types can satisfy this interface. The tilde symbol (~) signifies that `number` accepts any type with the same underlying type as `int`, `int64`, or `float64`. This means `Duration` and `usage` satisfy `number` because their underlying types are `int64` (Section 2.4.1) and `float64` (Listing 2.16), respectively.

Now, we can call the `avg` function to find the averages of different numeric types:

```
func main() {
    rts := []time.Duration{time.Second, 2 * time.Second}
    fmt.Println("average response time:", avg(rts))
    cpu := []usage{99.5, 50, 10.9}
    fmt.Println("average CPU usage    :", avg(cpu))
}
```

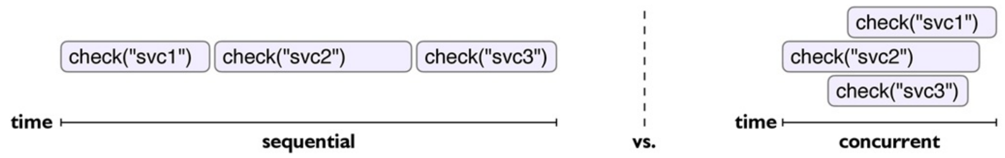
See the example in action in the link: https://go.dev/play/p/4cYpu7W0fSd

Generics are a new addition to Go (since version 1.18). We lived without them for more than ten years. We rarely needed them. The best practices are yet to be defined. So, we'll shy away from them in this book. We may cover them in greater detail, maybe in the second edition. You can learn more about generics at the link: https://go.dev/doc/tutorial/generics

2.5 Concurrency

Imagine the reliability team in our company manages hundreds of servers. They now want to learn about the fastest replica server to see the system's capacity better. Checking each replica's response time *sequentially* would take too long. To be more efficient and accurate, we'll *structure* our program to be concurrent. See Figure 2.17 for an illustration.

Figure 2.17 Sequential vs. concurrent approach of checking response times. The rectangle widths represent the time it takes to check a server's response time.



We illustrate two approaches: one is *sequential*, and the other is *concurrent*. Concurrency is doing many things at once, but not necessarily parallel: checks *may* start, run, and end at the same or at different times.

We can use *goroutines* (lightweight threads) to run *many* checks concurrently and a *channel* to safely transmit response times between goroutines without classic locks (i.e., mutexes).

Tip

Check out Rob Pike's talk on concurrency for a great intro: https://go.dev/blog/waza-talk. He explains why concurrency doesn't mean parallelism. Concurrency is a way of structuring a program, which *might* enable concurrent functions to run in parallel at runtime.

2.5.1 Goroutines

Before finding the fastest response time, let's discuss what goroutines are. For instance, we can run a goroutine by adding the `go` statement in front of a typical function like this:

```
func do(message string, t time.Duration) {
    for {            #A
        fmt.Println(message)
        time.Sleep(t)       #B
    }
}

func main() {
    go do("ping", time.Second)    #C
    go do("pong", time.Second)    #C
    select{}            #D
}
```

Note

The `go` keyword launches a new goroutine, running concurrently with others.

We might see the following once we run this program:

```
ping    #A
pong    #A
pong    #A
ping    #A
...
```

When a program starts, Go spawns an implicit *main goroutine* that runs the `main` function. Then, we run the `do` function concurrently in two separate goroutines. We use an empty `select` statement in the main goroutine to block indefinitely. Otherwise, the `main` function (hence the main goroutine) would return without waiting for the other goroutines to run.

Note

The garbage collector will automatically reclaim a goroutine when it returns.

2.5.2 Channels

Let's return to our goal of getting the fastest response. We'll use a channel to safely transmit response times between goroutines and get the fastest response time. For instance, we can create a channel that allow goroutines to transmit `time.Duration` values:

```
c := make(chan time.Duration)
```

The built-in `make` function initializes the channel. Since channels are complex and stateful types and managed by Go, `make` hands us the channel's pointer, which we save into `c`.

Note

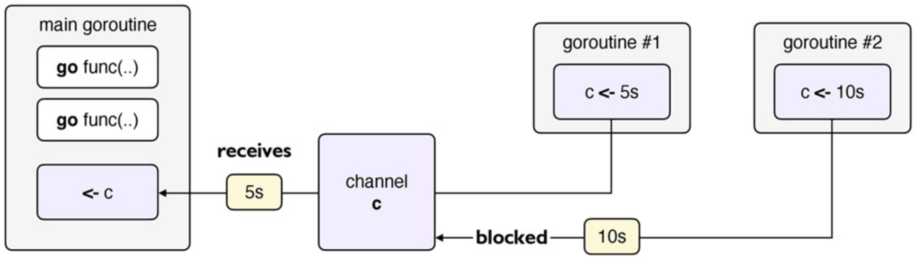
Channels can transmit any type of value. For instance, `chan int` is a channel that can transmit integers. Also, avoid passing around channels as pointers as they're already pointers.

Then, we can launch goroutines to concurrently send time durations to the channel. Lastly, we can get the fastest response from the goroutine that sends the message first. The arrow's placement around a channel determines if it's a *send* (`c<-`) or *receive* (`<-c`) operation.

```
func main() {
    c := make(chan time.Duration)        #A

    go func() {                #B
        time.Sleep(5 * time.Second)    #C
        c <- 5 * time.Second        #D
    }
    go func() {
        time.Sleep(10 * time.Second)
        c <- 10 * time.Second
    }

    firstResponse := <-c            #E
    fmt.Println(firstResponse)
}
```

This program will print 5 seconds when we run it. See Figure 2.18 for an illustration.

Figure 2.18 Goroutines run concurrently and send and receive over a channel. While the fastest goroutine's message goes through, the slowest gets blocked.



Let's see how our previous program works.

The main goroutine spawns two goroutines and shares a channel with them.

While the spawned goroutines send their timings, the main goroutine is *blocked* and waiting to receive the channel's first message. Since channels work like a FIFO (first-in, first-out) queue, the main goroutine receives the goroutine that sends the response time to the channel first (i.e., 5 seconds). This way, we can find the fastest response time.

Channels allow goroutines to synchronize and safely communicate with each other without data race issues. Whenever a goroutine tries to send or receive a message from a channel, the channel *blocks* that goroutine. This intentional blocking ensures that the message is *only received when a corresponding sent operation is ready*. We'll look at more examples soon.

Is it really the fastest response or the first message sent?

Typically, the first result received would likely be the fastest due to the concurrent execution, but this isn't a guarantee. To truly identify the fastest response, we need to implement additional logic that compares the timings received through the channel. But for simplicity and to focus on the mechanics of channels, we won't delve into that sorting logic here. We'll explore more advanced concurrency patterns later in the book.

2.5.3 Practice

With our knowledge of goroutines and channels, let's return to our goal of finding the fastest response time in a similar program to the previous one. See the next listing.

Listing 2.23 Collecting response times (con1/con.go)

```
package main

import (
    "fmt"
    "time"
    "math/rand/v2"
)

func main() {
    c := make(chan time.Duration)               #A

    go check("server1", c)               #B
    go check("server2", c)               #B

    fmt.Println("fastest response:", <-c)           #C
}

func check(url string, c chan<- time.Duration) {        #D
    d := rand.N(10 * time.Second)               #E

    fmt.Printf("checking %s will take %s...\n", url, d)
    <-time.After(d)                     #F
    c <- d                          #G
}
```

Once we make the channel, we launch goroutines and share the channel with them. Each runs `check`, which sends a duration to the channel after a random duration. To wait, we use `After`. It's similar to `Sleep` but `After` returns a channel that blocks its receivers until a duration elapses. We'll use this returned channel to wait for multiple channels soon.

```
func After(d time.Duration) <-chan time.Time    #A
```

While the spawned goroutines sleep, `main` waits to receive from the channel with `<-c`. Once a goroutine sends a duration to the channel with `c <- d`, `main` prints that duration.

Running this code, we grab the first message sent to the channel.

```
$ go run .
checking server2 will take 1s...
checking server1 will take 6s...
```

```
fastest response: 1s #A
```

While the first goroutines send a message in one second, the other one sends after six seconds. Since our program runs concurrently, it finishes in one rather than six seconds.

Receiving from the channel multiple times

Receiving from the channel for a second time would unblock the second goroutine. We might do that, perhaps, to find the average response times delivered by the goroutines:

```
func main() {
    // ...spawns goroutines...
    fmt.Println("average response:", (<-c + <-c) / 2)    #A
}
```

2.5.4 Goroutine leaks

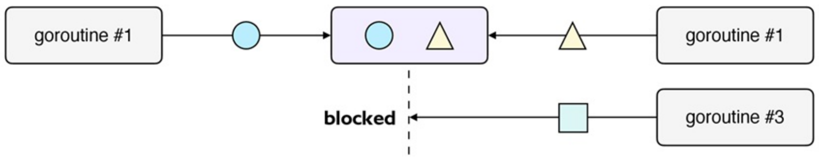
Channels block goroutines until corresponding receive and send operations. While the first goroutine in the previous program can send a value to the channel the slowest goroutine gets stuck because no other goroutine receives from the channel. This is a *goroutine leak*. A leaking goroutine never returns and unnecessarily consumes memory and CPU resources.

Note

Leaking a goroutine in our program isn't a real problem since all the goroutines are terminated when `main` returns. But it's crucial for long-running programs, like servers.

We can solve this problem using a *buffered channel*. Buffered channels have a queue and let goroutines send values without blocking. When the channel capacity becomes full, it acts exactly like a channel without capacity. Figure 2.19 shows a channel with a capacity of two.

Figure 2.19 A channel with a buffer capacity doesn't block goroutines until it's full.



Each goroutine sends a message, and the first two are buffered until a message is received on the other end, and the third goroutine to send will have to wait until that happens. In our case, we only have two goroutines (instead of three) and one of them leaks. So, we can use a channel with a buffer of two to fix the goroutine leak. See the next listing.

Listing 2.24 Using a buffered channel (con2/con.go)

```
func main() {
    c := make(chan time.Duration, 2)                #A
    ...
}
```

With this setup, two goroutines can send response times to the channel without getting blocked. This will fix the goroutine leak issue in the previous version in Listing 2.23.

2.5.5 Multiple channel operations

If the `check` operations take too long; users can feel like they're waiting forever. Instead, we can set a timer to end the program after, say, 5 seconds. To do that, we'll simultaneously listen to multiple channels using the `select` statement. One of them will be a timeout channel and the other will be a channel that receives `Duration` values. See the next listing.

Listing 2.25 Selecting multiple channels (con2/con.go)

```
func main() {
    timeout := make(chan struct{})            #A
    go func() {
        time.Sleep(5 * time.Second)
        close(timeout)                #B
    }()

    c := make(chan time.Duration, 2)
    go check("server1", c, timeout)       #C
    go check("server2", c, timeout)        #C

    select {
    case rt := <-c:                   #D
        fmt.Println("fastest response:", rt)
    case <-timeout:                   #E
        fmt.Println("timed out")
    }
}
```

The `select` statement lets a goroutine wait for multiple channels. If more than one channel is ready to receive (or send), it picks one randomly. This approach prevents faster goroutines from monopolizing channel operations, ensuring all have a fair chance to proceed. Go executes the next statement after the `select` statement once one of the channels becomes ready (unless we use another `select` that blocks inside the outer `select` statement).

Looking at the code, we first get our channels ready and spawn goroutines.

The first goroutine waits to signal with `close` after five seconds. Near the end of the `main` function, we use `select` to wait for multiple channels. If a check is finished first, we print the fastest time. If time runs out first, we print a "timed out".

We use a `timeout` channel here as *a signaling mechanism* rather than transmitting values. Closing the channel makes the main goroutine receive a signal from the `timeout` channel and stop listening to the channels. This way, we avoid indefinite blocking on the `c` channel.

Note

An empty struct (`struct{}`) doesn't occupy memory space and avoids copying while transferring values between channels when we're only interested in the signal itself.

Note

Close a channel for signaling purposes to unblock goroutines that are waiting to receive. Closing a channel makes it always ready to receive. Sending to a closed channel will cause panic at runtime. Avoid closing a channel for resource clean-up, as the garbage collector automates this process and automatically collects channels that are no longer referenced.

Never launch a goroutine without an exit strategy

You might notice in Listing 2.26 that we share the `timeout` channel with the `check` functions.

Imagine that our code in `main` is part of a long-running server. We use a combo of a `timeout` channel with `select`, which works well for timing out. That's half the battle. If the `check` function in a goroutine hangs, that goroutine won't stop, and our timeout won't help.

Let's fix this problem in the next listing.

Listing 2.26 Selecting multiple channels 2 (con2/con.go)

```
func check(url string, c chan<- time.Duration, timeout <-chan str
    d := rand.N(10 * time.Second)
    fmt.Printf("checking %s will take %s...\n", url, d)

    select {
    case <-time.After(d):                           #A
        c <- d                                  #A
    case <-timeout:                              #B
        // do nothing here as the function will return
    }
}
```

Similar to what we've done in `main`, we want to update the `check` function to wait for multiple channels. When the first goroutine closes the `timeout` channel, each goroutine that runs `check` will also get notified and return instead of sending a response time.

All goroutines wait either for a timeout or the other channel operation they do.

The main goroutine waits to receive, while others send a duration to the channel after sleeping for a random duration (blocking on the `After`'s channel). When the timer goroutine closes the channel, all our goroutines will call it a day because each will receive the closing signal. Here's what we get if we get a response within five seconds.

```
$ go run .
fastest response: 2s
```

Or, if timeout happens first:

```
$ go run .
timed out
```

This setup makes sure our program doesn't get stuck waiting for our goroutines to finish. Let's make a final change to simplify this program while preserving its behavior.

2.5.6 Timers to the rescue

Recall `After` from Section 2.5.3, which returns a channel that blocks its receiver until a time elapses. Since we're doing nothing but closing a channel in the previous timer goroutine, we can simply use `After` so we won't need to launch another goroutine. See the next listing.

Listing 2.27 Selecting multiple channels (con3/con.go)

```
// assuming check() is declared as in Listing 2.26

func main() {
    timeout := make(chan struct{})

    c := make(chan time.Duration, 2)
    go check("server1", c, timeout)
    go check("server2", c, timeout)

    select {
    case rt := <-c:
        fmt.Println("fastest response:", rt)
    case <-time.After(5 * time.Second):                   #A
        close(timeout)                           #B
        fmt.Println("timed out")
    }
}
```

We remove the excess goroutine and use `After` as a replacement. The second case in `select` will be selected if timeout happens first. Then, we'll `close` the `timeout` channel to notify the `check` goroutines to cancel their operations.

Running this will behave exactly like the code in the previous section. Using a timeout to *cancel* long-running operations is a good practice, and our program is now more robust. In practice, we use the `context` package to manage cancellation uniformly. More on that later.

2.5.7 Exercises

Note

Each exercise simulates doing the work by writing something on the console. You don't have to implement a real-world job queue system, etc.!

- Load Balancer: Distribute incoming requests to a pool of workers (goroutines). The workers should process any given task (e.g., "file compression", "generating a report", "sending an email", etc.). You may use interfaces to handle any task.
- Job Queue: Write a function that sends a task to a channel. Multiple goroutines receive from this channel and do the work. Use a buffered channel to limit the number of tasks that can be concurrently processed (e.g., 5 tasks at a time).
- Caching system: Goroutines handle set/get requests. Write functions to send caching and retrieval requests. The cache should be data race free. Use maps.
- Pub/Sub System: Simulate a simple publish/subscribe system using channels. Goroutines can subscribe to various topics and receive updates. Use maps.
- Heartbeat Check: Simulate a heartbeat check system where each goroutine represents a server. Use channels to send periodic heartbeat signals. The main goroutine should periodically print the average timings of the heartbeat signals.
- Create a scenario where goroutines have a time limit to complete a task. If they fail to complete the task in time, they should print a timeout message and return.

2.5.8 Wrap up

This section served as a primer on concurrency, giving us the tools to craft a concurrent code (we'll expand more in Chapter 7). Here's a wrap-up.

- Goroutines are lightweight threads that execute functions concurrently.
- Channels coordinate goroutines and safely transmit values between them.
- Unbuffered channels block until each sender pairs with a receiver.
- Buffered channels don't block senders until their capacity is full.
- The `select` statement lets goroutines wait on multiple channel operations.
- Closing a channel is for signaling and makes a channel always ready to receive.

2.6 Error handling

So far, we haven't used error handling to keep the examples to the point. Although we'll detail error handling throughout the book, let's briefly introduce it by looking at an example from C to compare Go's approach to handling errors. Check out the following C code.

```
file = fopen("error_handling_in_c.md", "r");
```

Languages like C blend errors with regular output, obscuring them. If `file` is `NULL`, an error occurred. If not, `file` holds a valid handle. This approach is called inband error handling.

In contrast, Go uses out-of-band error handling. Functions in Go can return multiple values, allowing us to return and check errors separately. This separation of concerns brings clarity. For instance, let's see the standard library's `Open` function, which returns a file and an error.

```
func Open(name string) (*File, error)    #A
```

This is the idiomatic way to return the `error` type (as a function's last result value).

2.6.1 Errors

The `error` type is a built-in interface type with a single `Error` method that returns a `string`:

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

Any type can implement the `Error` method to be an `error` type that we can return from functions as an error. For instance, the `net` package's `DNSError` is an `error` type.

```
type DNSError struct {
    Err    string // description of the error
    Name   string // name looked for
    Server string // server used
    // other fields
}

func (e *DNSError) Error() string   { .. }
func (e *DNSError) Temporary() bool { .. }
```

The `net` package uses this type to return detailed errors. We'll look at others later.

Using errors

The following example uses `Open` to get a file handle and error. If the returned `err` is `nil`, the returned `file` is safe for use; otherwise, `file` is `nil` and shouldn't be used.

```
func crawl() error {                   #A

    file, err := os.Open("urls.json")           #B
    if err != nil {                   #B
        return err // error occurred, don't use file    #B
    }
    // ...further processing, like crawling URLs...
    file.Close()
    return nil     // success
}
```

Note

A `nil` `error` means no error.

Now that we have a function that returns an `error`, we can check the error as follows. We call `crawl` and save the returned `error` value into the `err` variable (scoped to the `if` statement). We print the error if the error isn't `nil`. Otherwise, we print a success message.

```
// run this example at the link: https://go.dev/play/p/YWMlue4R-t
func main() {
    if err := crawl(); err != nil {            #A
        fmt.Println("error while crawling:", err)
        return
    }
    fmt.Println("crawling was successful")
}
```

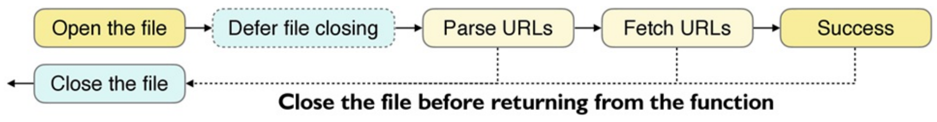
To sum up, in Go, *errors are values*. We return `nil` from a function when the function's operation is successful. Otherwise, an `error` value with an error message. This was an introduction, and we'll look into error handling in more detail in later chapters.

2.6.2 Deferring

Now that we're familiar with error handling in Go, there's a critical missing part in the puzzle: `defer`. Go lacks the usual *try-catch-finally* exception handling. Instead, we use `defer` within functions to *delay* certain actions until the function returns, usually for cleanup.

Imagine the previous `crawl` function does more work as in Figure 2.20. What if it gets and returns an error at any of these stages? We would have to close the file each time an error occurs! Instead, we can use `defer` to automatically close the file before `crawl` returns.

Figure 2.20 Making sure the file closes before the function returns.



Using `defer`, we can *ensure* closing the file if an error occurs at any stage while processing the file. The following example is deferring closing the file by registering the opened file's `Close` method. The function will defer calling `Close` until before the `crawl` function returns.

```
// See this example at the link: https://go.dev/play/p/ksLThfCxX2
func crawl() error {
    file, err := os.Open("urls.json")
    if err != nil {
        return err                  #A
    }
    defer file.Close()                  #B
    urls, err := parse(file)
    if err != nil {
        return err                  #C
    }
    // ...other operations that return an error...
    return nil                      #C
}
```

Opening a file can fail and return a `nil` file handle with an error. We should defer the file close only after we're sure the file has opened (this way is a common and idiomatic pattern). If we don't, and the file pointer is `nil`, the program will crash because we cannot call `Close()` on a `nil` `File` pointer. Also, arguments provided to a deferred function will be evaluated first without waiting for the outer function to return.

Note

It's also possible to defer with a closure to handle the error or do something else within the deferred function: `defer func() { err := file.Close(); /* handle it here */ }`

Note

Visit the link to learn about avoiding mistakes when using deferred functions: https://blog.learngoprogramming.com/gotchas-of-defer-in-go-1-8d070894cb01

Panic and recover

We can call `defer` multiple times and the deferred functions will run like a stack: last in first out. Go also has a `panic` function that works similarly to `throw` in other languages. There is also the `recover` function that enables us to handle panics. Still, we should use error values and not panic unless there is a truly unrecoverable issue. Although we'll use `defer` later in the book, here is more food for thought: https://go.dev/blog/defer-panic-and-recover.

2.6.3 Exercises

- Write a function that divides two numbers and returns an error if the denominator is zero. Handle this error in the `main` function that calls your function.
- Implement the `error` interface to create a custom error type that includes the time of the error occurrence. Use this in a function that generates an error.
- Open a file and use `defer` to ensure the file gets closed when the function returns. Additionally, check for an error when closing the file and print the error.
- Write a function with multiple `defer`s and observe the order in which they execute.

2.6.4 Wrap up

- Functions can return multiple values.
- Functions return `nil` `error` if their operations succeed. Otherwise, a non-nil error. Errors are values we can save into variables and return from functions.
- The `defer` statement runs a function before the surrounding function returns.

Leverage linters

Go doesn't warn us if we forget to check errors. For example, the `os` package's `Remove` function returns an error. But we might be accidentally ignoring checking the error:

```
os.Remove("secrets.json")
```

A linter, like "golangci-lint" can let us know when we forget to check the error:

```
$ golangci-lint run
main.go: Error return value of `os.Remove` is not checked (errche
```

You can download the linter from the link: https://golangci-lint.run.

2.7 Wrap up

This chapter introduced the basics of Go, arming us with all we need for the following chapters. Throughout the book, we'll detail and complement what we discussed in this chapter with complete real-world programs. The next chapter dives into testing.

2.8 Summary

- Package `main` can be compiled to an executable, while other packages provide code reusability by exporting their functionality to other packages.
- Everything is passed by value, including pointers.
- Pointers let us access values throughout without copying them.
- Arrays are fixed-length collections of elements of the same type.
- Slices are an underlying array's representation.
- Maps are key-value pairs.
- Go doesn't have classes, inheritance, or constructors.
- Structs bundle various types as fields in a single type.
- Methods are functions tied to a concrete type declared in the same package.
- Interfaces are implemented implicitly and abstract common behavior.
- Concurrency is a way to structure a program to deal with many tasks at once.
- Goroutines are lightweight threads that run concurrently.
- Channels are typed conduits for communication between goroutines.
- Errors are values that can be passed around, checked, and handled explicitly. The `error` type is an interface with a single `Error` method that returns a `string`.

- Deferring delays a function call until the surrounding function returns.

3 Idiomatic Testing

This chapter covers

- Writing and running tests using the tools provided by Go.
- Using table-driven testing and subtests to improve maintainability.
- Writing example tests to generate runnable documentation.

Automated tests ensure the code works today and will do so in the future.

We'll dive into testing after implementing a URL parser package called `url`. A simple replica of the standard library's `url` package with the same name. We'll keep this package simple enough so that we can focus more on the Go's testing features.

After learning the fundamentals of testing in Go, we'll explore table-driven tests to reduce duplication. While table-driven testing is helpful, it has some pitfalls, which we'll address using subtests.

In the next chapter, we'll explore measuring test coverage, benchmarking, optimization, and parallel testing.

Note

Find the source code of this chapter at the link:
https://github.com/inancgumus/gobyexample/tree/main/testing

3.1 Laying out the groundwork

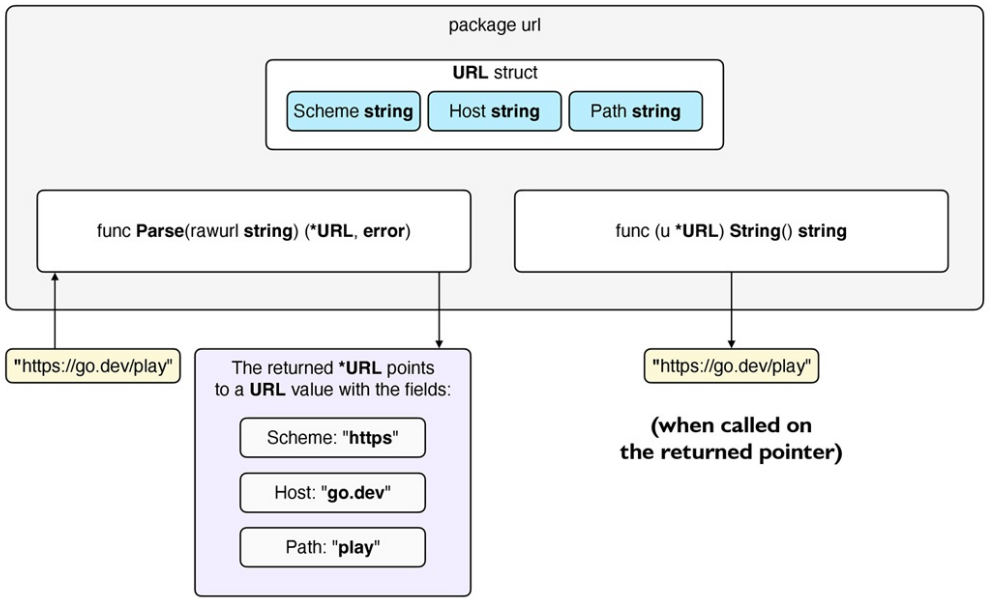
We're about to implement the `url` package. First, we'll explore our package's overview, discovering what it offers, how other packages can use it, and the idiomatic design decisions we make. Lastly, we'll discuss its implementation in the next sub-section.

Overview

Let's look at Figure 3.1 for an overview of the `url` package.

We have a `Parse` function that can parse a URL string and return a `URL` pointer and an `error`. The returned pointer points to a `URL` value (the pointer is not shown in the figure for brevity). A `URL` value's fields contain the parsed URL's parts, like `Scheme`, `Host`, and `Path`. We also have a `String` method to reassemble a URL string for easy logging.

Figure 3.1 The `url` package's overview. `Parse` returns a pointer to a `URL` value that contains the parsed URL's parts. `String` reassemblies the `URL` value as a URL string.


package url
URL struct
Scheme string
Host string
Path string
func Parse(rawurl string) (*URL, error)
func (u *URL) String() string
"https://go.dev/play"
The returned *URL points to a URL value with the fields:
Scheme: "https"
Host: "go.dev"
Path: "play"
"https://go.dev/play"
(when called on the returned pointer)

Now that we understand our package's overall structure, let's examine an example of its usage before diving into its implementation. This will help us better understand the package.

We'll parse a URL, handle the errors, and print the URL's parts and the exact URL again.

```
func main() {
    u, err := url.Parse("https://go.dev/play")
    if err != nil {
        log.Fatal(err)                         #A
    }
    fmt.Println("Scheme:", u.Scheme) // prints https
    fmt.Println("Host  :", u.Host)   // prints go.dev
    fmt.Println("Path  :", u.Path)   // prints play
    fmt.Println(u)                   // prints https://go.dev/pla
}
```

As discussed in the intro, we keep our package straightforward to focus on testing. Now that we know how others will use our package, let's discuss our design decisions.

Design decisions

For readability and clarity, the name of a package should be meaningful when its usage is combined with its items. A package's name is not just an accessor for a package's items. It makes it straightforward to understand what each exported package item does.

Tip

A package's name concisely communicates what it offers, not what it does. For instance, the `url` package offers a way to work with URLs, so we call it `url` rather than `url_parser`. In the future, we might add more functionality and won't need to change the package's name.

For instance, other packages can call `Parse` like this: `url.Parse`. This makes it effortless to grasp what `Parse` does: parses a URL. Without `url`, it would be difficult to understand what `Parse` does. However, if `Parse`'s name were `ParseURL`, it would unnecessarily stutter: `url.ParseURL`. For clarity, it's idiomatic to avoid stuttering when naming package items.

Note

Conventions are nice until they face the real world. For instance, the standard library also has a `url.URL` type. Initially, this type belonged to the `http` package. Due to circular import issues, the Go team has moved the type to the `url` package, resulting in the repetition.

Other than naming, it's idiomatic to implement the standard library's interfaces where we could. For instance, here we see implicit interfaces in action: `Println` can nicely print a `*URL`, even though it's a struct pointer. It works because `String` satisfies `fmt.Stringer`:

```
package fmt


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

Lastly, let's discuss why we don't use getters and setters for the `URL` fields. Although we could add accessors using methods, Go doesn't have built-in support for accessor methods. For instance, other packages can directly access a `URL` value's fields because we export `Scheme`, `Host`, and `Path` fields. While this might be a big no in other languages, we don't excessively use methods for getting and setting every struct field in Go.

Note

We could hide these fields by not exporting them like `scheme`, `host`, and `path`. Then, exported methods like `Scheme()`, `SetScheme(s string)`, etc., could get and set these fields.

We might use methods if we have business logic to implement, protect a value's internals, or implement an interface. Methods might also be helpful for making a type immutable by hiding its fields and returning new values after a change. To wrap up, methods are tremendously helpful unless we overuse them. We'll keep expanding idiomatic ways of working with Go throughout the book.

Implementing the package

Now that we're familiar with the `url` package, let's implement the first version in the next listing. In this version, `Parse` returns a hard-coded value and `nil` (always succeeds). `String`, on the other hand, works as expected for the majority of cases (we'll see where it fails later).

Listing 3.1 The url package (url/url.go)

```
package url

import "fmt"

// A URL represents a parsed URL.
type URL struct {
    Scheme string
    Host   string
    Path   string
}

// Parse parses a raw url into a URL structure.
func Parse(rawURL string) (*URL, error) {
    u := &URL{                    #A
        Scheme: "https",
        Host:   "go.dev",
        Path:   "play",
    }

    return u, nil                     #B
}

// String reassembles the URL into a URL string.
func (u *URL) String() string {
    return fmt.Sprintf("%s://%s/%s", u.Scheme, u.Host, u.Path)
}
```

This version serves as a starter so we can dive into testing rather than focusing on implementation details. We'll improve this code until it can parse URLs. Our tests will greatly help us while doing so and let us know whether we're implementing the correct functionality.

3.2 Testing

Go comes with built-in testing support and we don't have to install third-party tools. We can use the following components to write and automatically run our tests and verify our code.

- The `go test` tool—A command-line utility to run tests and show their results.
- The `testing` package—A set of functions and types for writing tests.

Testing in Go isn't much different from writing Go code. Tests are written inside functions in usual Go source files with specific conventions:

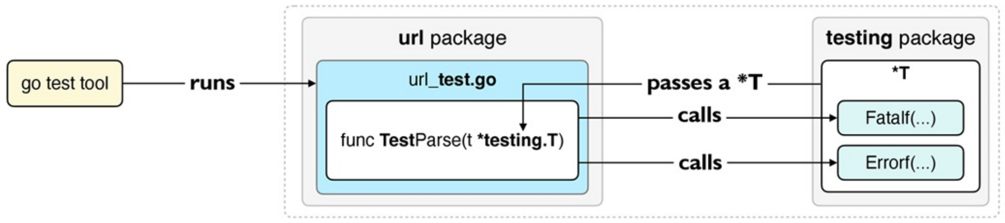
- Test file names end with `_test.go`.
- Test functions begin with `Test` and take a `*testing.T` input, usually named `t`.

Remember

From now on, we'll say `*T` while referring to `*testing.T` for brevity.

Figure 3.2 shows a test function for the `Parse` function. Since we want to test the `url` package's `Parse`, we name the test file `url_test.go` and the function `TestParse`.

Figure 3.2 The go test tool and the `testing` package collaborate in running tests. Tests get a `*T`, which has methods for tests to call to log, report failures, etc. Although we have one for each here, there can be multiple test files and functions.



The `*T` is a struct type with test-related methods attached. We can use `*T`'s methods from our test functions to report results, print logs, etc. For instance, `Fatalf` marks a test as failed and stops its execution, which is ideal when a failure is so critical that no further testing makes sense. `Errorf`, on the other hand, logs a failure but lets the test continue.

Now that we're familiar with testing, let's test the `url` package.

Note

While both `Fatalf` and `Errorf` mark a test as failed, `Fatalf` stops the test's run. Since tests are independent, a `Fatalf` call in a test won't stop the other tests and they keep running.

3.2.1 Writing tests

We're about to test the `url` package's `Parse` function. Check out the test function `TestParse` in the next listing. The test receives `t` of type `*T` and report failures either with `t.Fatalf` or `t.Errorf`. We'll look at other methods the `*T` type offers later.

Listing 3.2 Testing the url package (url/url_test.go)

```go
package url


import "testing"                                #A

func TestParse(t *testing.T) {
    const uri = "https://go.dev/play"

    got, err := Parse(uri)                      #B
    if err != nil {
        t.Fatalf("Parse(%q) err = %q, want <nil>", uri, err)
    }
    want := &URL{                               #D
        Scheme: "https", Host: "go.dev", Path: "play",
    }
    if *got != *want {                          #E
        t.Errorf("Parse(%q)\ngot  %#v\nwant %#v", uri, got, want)
    }
}
```

Returning to our test, we first declare a constant `uri`, the target URL string we want to parse. We then call `Parse`, which returns a `*URL` and an `error`. If `Parse` fails, we report a failure and halt the test with `Fatalf` since there's no point in proceeding with an invalid URL.

Once we confirm the parsing is successful, we compare the actual result (`got`) against our expected result (`want`). Then, we use `Errorf` to report a failure if the results don't match. Even if `Parse` gives us something unexpected, the test can still carry on.

Warning

A `nil` `*URL` from `Parse` would cause a panic if we tried to dereference it. That's why we use `Fatalf` to stop the test from running the lines where we dereference the pointers.

Now that we better understand writing a test function, let's look at another example. The next listing demonstrates writing a test function `TestURLString` to test the `url` package's `String` method. We verify if `String` can reassembly a `*URL` into a string.

Listing 3.3 Testing the url package (url/url_test.go)

```
func TestURLString(t *testing.T) {

    u := &URL{
        Scheme: "https",
        Host:   "go.dev",
        Path:   "play",
    }

    got  := u.String()
    want := "https://go.dev/play"
    if got != want {
        t.Errorf("String() = %q, want %q", got, want)
    }
}
```

Understanding when to use `Fatalf` versus `Errorf` is crucial for effective tests. `Fatalf` is best for errors that invalidate the rest of the test, as it stops execution immediately. For other cases, we should prioritize using `Errorf`. This way, the test can yield more information even after a failure, and we can have a complete picture of all issues without rerunning tests.

Tip

It's idiomatic to name tests after what they test: `TestParse` for `Parse` and `TestURLString` for `String`. The latter avoids conflicts with the tests of other types with a `String` method.

Running tests

Now that we have tests, we can run them using the `go test` tool. The test tool is versatile and lets us run tests in many ways. Let's first look at the simplest way to run tests.

```
# remember running the following while in the testing/url directo
```

```
$ go test
PASS
ok      github.com/inancgumus/gobyexample/testing/url      0.262
```

The `go test` tool compiles the tests, tells the `testing` package to run them, shows the results, and throws away the compiled test binary. Here, our tests pass and take ~0.25 seconds to run. Rerunning them will take less time since the test tool caches the results.

We know our tests pass but we don't know which ones run. Let's turn on the *verbose* flag (`-v`) to ensure they all run. Tests can also output logs when this flag is set.

```
$ go test -v

--- PASS: TestParse
--- PASS: TestURLString
```

Moreover, we can shuffle tests to make sure that each test is isolated. Tests depending on others' success or failure will make it difficult to find the root cause of potential problems.

```
$ go test -shuffle=on

--- PASS: TestURLString
--- PASS: TestParse
```

We can see that tests are shuffled if we compare them to the previous run above.

We'll look at other interesting ways of using the `go test` later.

Note

When building our code using `go build` or `go run`, the compiler disregards the tests; they leave no footprint on the final binary size. Tests are considered only by the `go test` tool.

3.2.2 Writing a failing test

We verified our code works for a single URL. We now want to test `Parse` with additional URLs to make sure it also parses those URLs. The next listing shows adding another test to verify `Parse` works with a different URL. We know that it does not (recall that from Listing 3.1).

Listing 3.4 Testing the Parse function #2 (url/url_test.go)

```
func TestParseWithoutPath(t *testing.T) {
    const uri = "https://github.com"

    got, err := Parse(uri)
    if err != nil {
        t.Fatalf("Parse(%q) err = %q, want <nil>", uri, err)
    }

    want := &URL{
        Scheme: "https", Host: "github.com", Path: "",
    }
    if *got != *want {                       #A
        t.Errorf("Parse(%q)\ngot  %#v\nwant %#v", uri, got, want)
    }
}
```

Running the tests will report a failure. We demonstrate the failure message here with added newlines and space to make it more readable. Check out the sidebar ("Comparing complex values and printing diffs") for an alternative way.

```
$ go test
--- PASS: TestParse
--- FAIL: TestParseWithoutPath
    url_test.go: Parse("https://github.com")                 #A
    got  &url.URL{Scheme: "https", Host: "go.dev",     Path: "pla
    want &url.URL{Scheme: "https", Host: "github.com", Path: ""}
--- PASS: TestURLString
```

Let's now discuss what this message tells us about the test failure and its usefulness.

Note

We're starting to duplicate the test logic in our parsing tests. They're identical except that they use different inputs and outputs. We'll address this issue in the next section.

3.2.3 Writing descriptive test failure messages

The previous failure message showed us `Parse` was given a URL but incorrectly parsed it. The bright side is we got a descriptive message to diagnose the issue quickly, which is crucial for effective debugging.

```
Here's the failure message format we use to see what's wrong inst
t.Errorf("Parse(%q)\ngot  %#v\nwant %#v", uri, got, want)
```

This structured *"got; want"* convention provides a clear snapshot of what we *get* versus what we *want*. It helps us quickly identify what part of the code failed without looking at the tested code. This is a real-time-saver when working with large codebases or many tests.

Comparing complex values and printing diffs

Comparing struct values is not always trivial. One way is using the `reflect` package's `DeepEqual` method if we have a complex type with many nested fields and pointers to follow.

A much better way is to utilize `go-cmp`. We can get the package as follows.

```
$ go get github.com/google/go-cmp/cmp
```

Then, we can import the `cmp` package in a test file and compare values like this.

```
if diff := cmp.Diff(want, got); diff != "" {
    t.Errorf("Parse(%q) mismatch (-want +got):\n%s", uri, diff)
}
```

The failure message will look like this after running the test (it's even customizable).

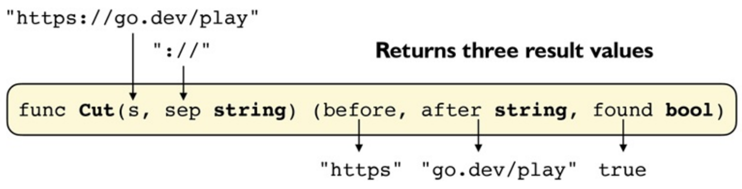
```
Parse("https://github.com/inancgumus") mismatch (-want +got):
    - Host: "github.com", + Host: "go.dev",
    - Path: "inancgumus", + Path: "play",
```

Check out its documentation: pkg.go.dev/github.com/google/go-cmp/cmp

3.2.4 Fixing the code

Before leaving this section, let's fix `Parse`. We'll use the `strings` package's `Cut` function to parse a URL. Figure 3.3 shows how `Cut` works. This function splits a string using a separator, yielding the portions `before` and `after`. It also tells us if the separator is `found` within the string. It returns the whole string in `before` if the separator isn't found.

Figure 3.3 The function divides a string, returning segments `before` and `after` the separator. The `found` result value indicates the detection of the separator.


"https://go.dev/play"
"://"
Returns three result values
func Cut(s, sep string) (before, after string, found bool)
"https"
"go.dev/play"
true

Let's use `Cut` within `Parse` to parse a URL. See the next listing.

Listing 3.5 Fixing the Parse function (url/url.go)

```
func Parse(rawURL string) (*URL, error) {                #A
    scheme, rest, ok := strings.Cut(rawURL, "://")            #B
    if !ok {
        return nil, errors.New("missing scheme")            #C
    }

    host, path, _ := strings.Cut(rest, "/")

    u := &URL{
        Scheme: scheme,
        Host:   host,
        Path:   path,
    }

    return u, nil
}
```

We start with parsing the `scheme`. The parsing stops and returns an `error` if the URL lacks a scheme. Otherwise, we continue and parse the `host` and `path`. Lastly, we inject what we parse into a `URL` value, returning a pointer to this value and a `nil` error (success).

We use the `errors` package's `New` function to return an `error` value that contains the "missing scheme" message. Callers of `Parse` can see this error message if they want to print the error. We'll look into other alternatives for handling errors in the upcoming chapters.

This fix was straightforward. Let's rerun our tests and see if they pass or fail.

```
$ go test -v
PASS: TestParse...PASS: TestParseWithoutPath...PASS: TestURLStrin
```

Tests pass. We could also write a test function that verifies if `Parse` returns an error when it gets a URL string that lacks a scheme. Let's leave this as an exercise for you.

Remember

The `errors.New` function returns a new `error` value. The `error` is an interface type with a single `Error() string` method. Since errors are interface values, we can provide our own `error` implementation. In practice, we often create errors using the standard library's `errors.New` and `fmt.Errorf` functions. We'll see examples of the latter later.

Dive deep: Using named result values for readability

`Cut` names its result values to enhance readability since each is a primitive type and isn't clear enough without a name: `string`, `string`, and `bool`. Deciphering the function's purpose would be challenging without using named results. Still, we should be careful. Each named result value becomes a local variable within the function and might introduce subtle bugs. We should name results if they provide clarity, not for the sake of using them as variables.

3.2.5 Wrap up

We're now familiar with the `go test` tool and the `testing` package. With this introductory knowledge, we're ready to dive deep into testing in Go. In the next section, we'll explore table-driven testing to reduce duplication in our parsing tests. Let's wrap up.

- The `go test` tool runs tests.
- Test files have a `_test.go` suffix. Tests have a `Test` prefix and take a `*T`.
- `Errorf` and `Fatalf` make a test fail. The latter also stops the test's run. Using `Errorf` instead of `Fatalf` helps us see the whole picture without rerunning tests.
- Descriptive failure messages help us quickly spot the failure reasons.

3.3 Table-driven testing

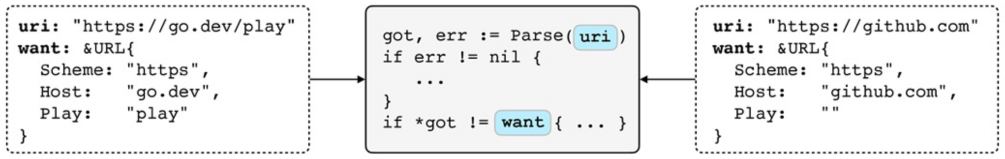
Earlier, we crafted two tests to test `Parse`. Looking at them in Listing 3.2 and 2.4 might be helpful before going on. These tests have identical testing logic. Instead of this duplication for each URL we want to test, the idiomatic approach uses table-driven testing, which separates the test data from logic and reuses that same logic to improve maintainability.

Remember

Table-driven tests separate the test data and logic to improve test maintainability.

Figure 3.4 shows how we reuse the test logic shared by our earlier tests. In this new approach, we run and test each test case (data) using the same test logic. Here, the `uri` field is the URL string we pass to `Parse`, while `want` is what we expect `Parse` to return.

Figure 3.4 Using the same test logic when testing with various data.



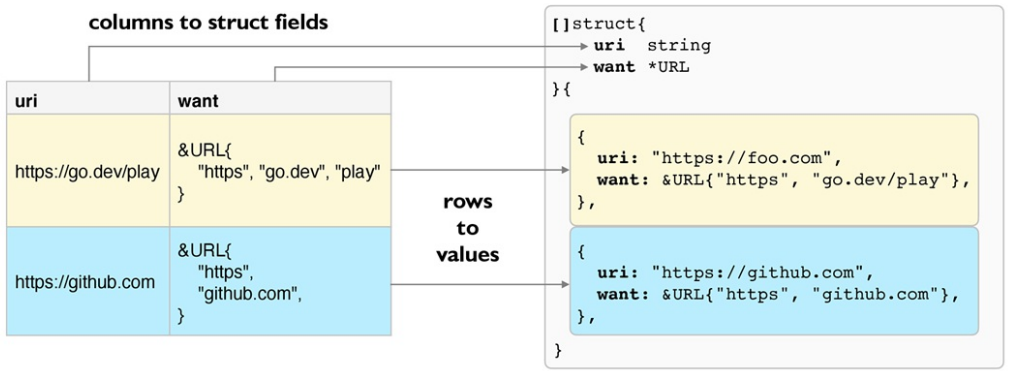
```
uri: "https://go.dev/play"
want: &URL{
  Scheme: "https",
  Host:   "go.dev",
  Play:   "play"
}
```

```
got, err := Parse(uri)
if err != nil {
   ...
}
if *got != want { ... }
```

```
uri: "https://github.com"
want: &URL{
  Scheme: "https",
  Host:   "github.com",
  Play:   ""
}
```

Check out Figure 3.5 for another way to look at this. We have a table for our test cases and the corresponding Go code: a slice of anonymous struct values that maps test cases. The same test logic can use these slice elements to test `Parse` from various angles.

Figure 3.5 Illustration of mapping a table of test cases to a slice of struct values.



uri	want
https://go.dev/play	&URL{ "https", "go.dev", "play" }
https://github.com	&URL{ "https", "github.com", }

```
[]struct{
    uri  string
    want *URL
}{
  {
    uri: "https://foo.com",
    want: &URL{"https", "go.dev/play"},
  },
  {
    uri: "https://github.com",
    want: &URL{"https", "github.com"},
  },
}
```

The figure's test cases match the data our previous parsing tests used. We'll put the same data into an anonymous struct slice and use the test logic once in a single test.

Although we currently have two test cases, adding more in the future is straightforward. We can use the same test logic to reduce duplication and pave the way for increased test coverage (which we'll dive into in the next chapter). Crucial benefits that we can't overlook.

Note

We're omitting testing the edge cases to keep the examples clear.

3.3.1 Creating a table of test cases

Now that we're familiar with table-driven tests, let's create a table of test cases. Listing 3.6 shows the declaration of a package-level anonymous struct variable named `parseTests`.

Note

Package variables stay alive for the entire program execution and are never garbage collected. Still, the variables defined by tests are only alive during testing.

We first define the struct slice type with the `name`, `uri`, and `want` fields.

Since we'll run each test case under the same test function, the `name` field will be helpful for logging the currently running test case. The `uri` field is the URL string, and `want` is the expected result. We'll pass `uri` to `Parse` and test if it returns the expected URL.

After defining the type, we fill out the slice with the test cases. We're adding the data that the previous `TestParse` and `TestParseWithoutPath` used.

Listing 3.6 A table-driven test (url/url_test.go)

```
var parseTests = []struct {                    #A

    name string                                #A
    uri  string                                #A
    want *URL                                #A
}{
    {                               #B
        name: "full",                              #B
        uri:  "https://go.dev/play",                      #B
        want: &URL{Scheme: "https", Host: "go.dev", Path: "play"}
    },                               #B
    {                               #C
        name: "without_path",                          #C
        uri:  "http://github.com",                          #C
        want: &URL{Scheme: "http", Host: "github.com", Path: ""},
    },                               #C
    /* many more test cases can be easily added */
}
```

We declared a *package variable* (think global) that the test functions in the `url` package can use. Sharing the test cases lets us use them from different tests when needed. For instance, we'll use these test cases when we're upgrading our test function to subtests.

Another bright side is this variable is only available to tests; non-test code cannot reach it. This is because test code is only considered while running tests.

Remember

Table-driven tests make it easy to add new test cases. For instance, when a bug is found, we can simply add another test case without changing the rest of the test code.

3.3.2 Writing a table-driven test

Now that we have a slice of test cases, let's implement our test logic in the next listing.

Listing 3.7 A table-driven test (url/url_test.go)

```
func TestParseTable(t *testing.T) {
    for _, tt := range parseTests {                     #A
        t.Logf("run %s", tt.name)                         #B

        got, err := Parse(tt.uri)                         #C
        if err != nil {                         #C
            t.Fatalf("Parse(%q) err = %v, want <nil>", tt.uri, er
        }                             #C
        if *got != *tt.want {                         #C
            t.Errorf("Parse(%q)\ngot  %#v\nwant %#v", tt.uri, got
        }
    }
}
```

We loop over test cases and run the same test logic for each test case. The `tt` loop variable contains the next test case's data. Using `Logf` makes it easier to see which test case we're running. This test is similar to the previous parsing tests. The most significant difference is that the test logic fetches data from a slice of structs. Now, let's give this test a run.

Exercise

Write a table-driven test for the `String` method. You can put the test cases in the same test rather than in a package-level variable as you won't share them with other tests.

Nitpick: About the test's name

We use the `TestParseTable` name for illustration purposes. In practice, we would just call it `TestParse`. Picking distinct names for each test helps us easily refer to them in the book.

3.3.3 Running a specific test

We can use the `run` flag to execute only `TestParseTable` to focus on this specific test.

```
$ go test -v -run=TestParseTable$        #A

=== RUN   TestParseTable
    run full
    run without_path
--- PASS: TestParseTable
```

The test has succeeded. However, unlike our previous tests, which ran each case independently, `TestParseTable` now handles multiple test cases together (the same test runs them). This consolidates our tests and reduces duplication but at the expense of the isolation we had before. We need to assess the implications of this trade-off.

Tip

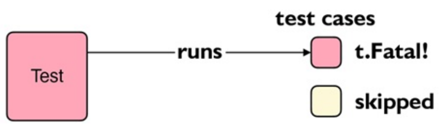
The `run` flag runs only the specified tests and supports regular expressions using the RE2 syntax. For more info, visit: github.com/google/re2/wiki/Syntax

3.3.4 Issues with table-driven tests

We have two outstanding shortcomings with our current table-driven test setup.

Figure 3.6 shows the first issue: a fatal failure (i.e., `Fatalf`) causes the test to skip running the remaining test cases. When each test case was independent, a failure in one wouldn't affect the others. But, we now run all test cases in a single test, and a fatal failure stops all.

Figure 3.6 Skips the other case because of a fatal failure while running the first.



Let's reveal the issue with an example. The next listing adds another test case at the start of our table. It's a valid URL with a *data scheme* containing a Base64 encoded text. Although the scheme exists, our parser won't parse this URL and will return a "missing scheme" error.

Listing 3.8 Inserting a data scheme test case (url/url_test.go)

```
var parseTests = []struct { ... }{
    {
        name: "with_data_scheme",
        uri:  "data:text/plain;base64,R28gYnkgRXhhbXBsZQ==",
        want: &URL{Scheme: "data"},
    },
    { name: "full",         /* ..rest is omitted... */ },
    { name: "without_path", /* ..rest is omitted... */ },
}
```

This test case should cause our test to call `Fatalf` and skip the remaining test cases.

```
$ go test -v -run=TestParseTable$
=== RUN   TestParseTable
    url_test.go:77: run with_data_scheme
    url_test.go:81: Parse("data:<omitted>") err = missing scheme,
--- FAIL: TestParseTable
```

Our test calls `Fatalf` during the data test case step, causing the remaining test cases to be skipped. Another issue, perhaps a less significant one in our case, is that, unlike test functions, we can't selectively run a specific test case using the `run` flag. This is because we have a single test. Luckily, as we'll see, we can solve these issues using subtests.

Note

Unlike tests that run a single test case, table-driven tests run every case in a single test.

Dive deep: Fail fast

Sometimes, stopping tests at the first sign of trouble can streamline debugging. For this purpose, I've contributed the `failfast` flag, released in Go 1.10. It halts further testing upon the first reported test failure, letting us focus our attention where needed most.

Here's the usual test run without `failfast`:

```
$ go test
--- FAIL: TestParseTable
--- PASS: TestURLString
```

And here it is with `failfast` enabled, stopping early:

```
$ go test -v -failfast
--- FAIL: TestParseTable
```

The choice between halting immediately or after all failures depend on whether we're isolating a specific issue or assessing the overall health of the code.

3.3.5 Wrap up

We're now familiar with table-driven testing. Let's wrap up.

- Test cases are scenarios with the input and output data we want to test.
- Table-driven tests reuse the same test logic, reducing duplication and helping to achieve better maintainability and increased test coverage.

However, there are also downsides when using table-driven testing:

- Test cases cannot be run individually as the same test runs all.
- A fatal test failure causes the skipping of remaining test cases.

3.4 Subtests

Earlier, we crafted a table-driven test and learned that fatal failures while running test cases can skip running the remaining ones. Luckily, subtests solve this downside of table-driven tests: subtests are similar to test functions and enable us to run test cases independently.

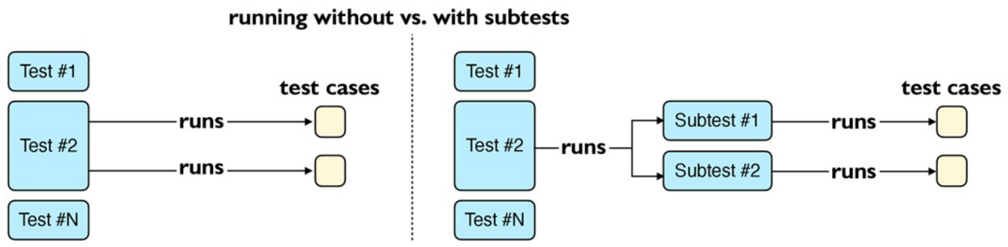
Remember

Subtests can fail without stopping others.

3.4.1 Understanding subtests

Let's compare the table-driven and subtest approaches in Figure 3.7. The left-hand side shows our early approach: the same test runs test cases. On the right, we use subtests. The test runs and splits into two subtests, each separately running a test case.

Figure 3.7 Running test cases within the same test vs. in separate subtests.



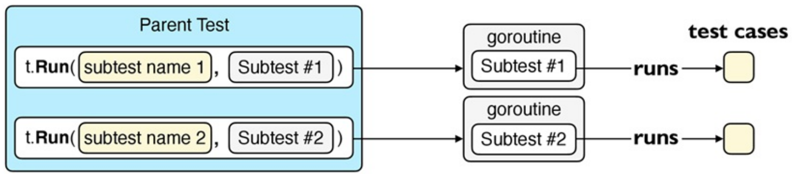
We can run a subtest using the `*T` type's `Run` method. Figure 3.8 shows a test that calls `Run` to launch two subtests. Each subtest is running a test case in a separate goroutine.

Remember

Like tests, subtests have a name and their own test logic and run in a separate goroutine. This isolates tests and lets us optionally run them in parallel to boost performance.

Recall from the previous chapter that goroutines run a function concurrently. For the `Run` method to run a test case in a separate goroutine, we must pass it a function. This function will contain test logic and it will run a test case from our test cases table.

Figure 3.8 Test runs two subtests, each running a test case in a separate subtest.


Parent Test
t.Run(subtest name 1 , Subtest #1)
t.Run(subtest name 2 , Subtest #2)
goroutine
Subtest #1
goroutine
Subtest #2
runs
runs
test cases

To wrap up, subtests are test functions run separately and are impervious to the fatal failings of others. Now that we're familiar with subtests, let's dive into using them. We'll iterate over our table-driven tests, running each test case by a subtest to bolster test resiliency.

3.4.2 Writing a table-driven test with subtests

Let's put subtests into practice in the next listing. Like our previous test, we loop over our test cases. On each iteration, we call the `Run` method to run a subtest. Each subtest gets a unique name and a *closure* that wraps the test logic that runs a test case.

Listing 3.9 Table-driven subtests (url/url_test.go)

```
var parseTests = []struct { ... }{
    { name: "with_data_scheme", ... },
    { name: "full",             ... },
    { name: "without_path",     ... },
}

func TestParseSubtests(t *testing.T) {
    // common test setup and teardown logic can be put here
    for _, tt := range parseTests {
        t.Run(tt.name, func(t *testing.T) {                #A
            got, err := Parse(tt.uri)
            if err != nil {
                t.Fatalf("Parse(%q) err = %v, want <nil>", tt.uri
            }
            if *got != *tt.want {
                t.Errorf("Parse(%q)\ngot  %#v\nwant %#v", tt.uri,
            }
        })
    }
}
```

Note

Closures are anonymous functions that can access their surroundings, like variables.

The closure takes a `*T` so we can call `Fatalf` and `Errorf` only to report failure for the current subtest. Since closures can access their surroundings, we can use `tt.uri` and `tt.want`. Remember that this `*T` that the closure takes is a different `*T` of its parent test. We'll discuss more about this before wrapping up this subsection.

Our test has become more robust. Although the code inside each subtest's closure remains logically the same as our previous table-driven test, subtests run independently. We can run and examine more tests even in the presence of some fatal failures.

Even if one subtest fails with `Fatalf`, others will go on. This is because each `T` pointer represents a single test. When `Run` executes, it passes a new `T` pointer (`t` in the code) to the subtest's closure. This allows for more granular control over test execution and failure handling. Each subtest can call `Fatalf` without failing others because the `T` pointer is unique per subtest. This means we should never share it with other tests.

Warning

Never share a test's (including subtests) `*T` with another test. Doing so could lead to unpredictable behavior and compromise the validity of test results.

Lastly, we see a comment at the beginning of the test related to common test setup and teardown logic. Since a test function runs subtests, using subtests makes it straightforward to share a common setup and teardown logic with subtests: we could open a connection to a database and close it when all subtests finish. Chapter 11 will delve into this.

Dive deep: Closures and memory management

Closures are function values and may hold pointers to the variables they reference. The garbage collector cannot reclaim the memory for these variables as long as the closure is in scope, shared, or hasn't returned. We should be mindful of their lifetime to prevent leaks.

3.4.3 Running subtests

Remember from Section 3.3.4 that our table-driven test wasn't running the rest of the test cases when a prior one failed. Now that we're using subtests, let's see the difference. The parent test `TestParseSubtests` will `Run` each subtest sequentially and wait for them.

```
$ go test -v -run=TestParseSubtests$

--- FAIL: TestParseSubtests
    --- FAIL: TestParseSubtests/with_data_scheme
    --- PASS: TestParseSubtests/full
    --- PASS: TestParseSubtests/without_path
```

Thanks to our new subtest approach the rest of the test cases run even if one of them fails. Here, the first subtest fails, leading to the parent test's failure. Still, the parent test continues to run the rest of the subtests as each subtest is independently run.

Moreover, since subtests have a name, we see their names in the test report, which enhances readability and maintainability, as they allow us to understand the test's purpose at a glance. Moreover, we can use subtest names to run individual subtests selectively. We can do so because each subtest is a separate test function.

```
$ go test -v -run=TestParseSubtests/full$

--- PASS: TestParseSubtests
    --- PASS: TestParseSubtests/full
```

This time, the parent test passed since the subtest also passed. Although the parent test calls `Run` for all our subtests to run them, the `testing` package intentionally ignores them.

3.4.4 Fixing the code

We still have a failing subtest:

```
$ go test -run=TestParseSubtests -failfast        #A

=== RUN   TestParseSubtests/with_data_scheme
    Parse("data:text/plain;<omitted>") err = missing scheme, want
```

To fix this issue, we'll extend the `Parse` function's capabilities to interpret opaque and typical URLs schemes. Let's look at the solution in the next listing.

Note

We consider a URL opaque if the part after the scheme doesn't contain a slash.

Listing 3.10 Fixing the Parse function (url/url.go)

```
func Parse(rawURL string) (*URL, error) {
    scheme, rest, ok := strings.Cut(rawURL, ":")        #A
    if !ok {
        return nil, errors.New("missing scheme")
    }

    if !strings.HasPrefix(rest, "//") {             #B
        return &URL{Scheme: scheme}, nil
    }

    host, path, _ := strings.Cut(rest[2:], "/")        #C

    u := &URL{
        Scheme: scheme,
        Host:   host,
        Path:   path,
    }

    return u, nil
}
```

Instead of cutting URLs at "//", we now do it at ":" to be compatible with opaque and typical URLs. Let's see what our variables would look like for both URL schemes:

- Opaque: `rawURL`="data:text/plain", `scheme`="data", `rest`="text/plain"
- Typical: `rawURL`="https://go.dev/play", `scheme`="https",

`rest`="//go.dev/play"

Once the scheme is parsed, `Parse` now stops if it encounters an opaque URL, returning a `URL` with only the scheme. Otherwise, it proceeds to parse the host and path from `rest[2:]`. This is called *string slicing* and omits the first two bytes: e.g., "*//go.dev*" becomes "*go.dev*".

Remember

Strings can be sliced because strings are inherently byte slices.

After adjusting `Parse`, we should rerun our tests to verify the fix.

```
$ go test -v -run=TestParseSubtests$

--- PASS: TestParseSubtests (0.00s)
    --- PASS: TestParseSubtests/with_data_scheme
    --- PASS: TestParseSubtests/full
    --- PASS: TestParseSubtests/without_path
```

With these changes, `Parse` can correctly interpret newer and traditional URL formats. Although this is good progress towards a robust parser, our focus will remain on testing in Go rather than creating a full-fledged parser that can handle any URL format.

Note

Recall from the previous chapter that slices are views. Slicing returns a new slice that sees the whole or some part of the underlying array. Similarly, slicing a string returns a string without copying the underlying bytes data. Also, `rest[2:]` is a shortcut for typing `rest[2:len(rest)]`. Similarly, there is `rest[:2]` which is a shortcut for typing `rest[0:2]`.

Warning

We should be careful while slicing strings since the underlying string might be Unicode encoded. Slicing a Unicode string might be dangerous because a single character can span into multiple bytes. See the link for more details: https://go.dev/blog/strings.

3.4.5 Wrap up

- Subtests are standalone test functions run by another test function.
- A test function can run a subtest using `*T`'s `Run` method.
- Subtests can fail with a fatal failure without stopping others.
- Combining subtests with table-driven tests is powerful for resilient testing.
- Sharing a test's `*T` with another test can cause unpredictable results.

Dive deep: Top-level tests are subtests, too.

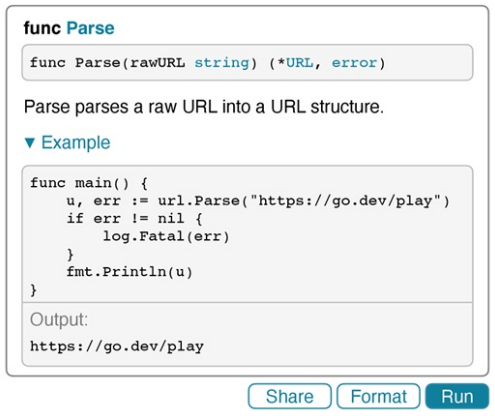
To prevent confusion and distraction, we didn't explain that tests are also subtests. Internally, the `testing` package runs each top-level test (whose names start with Test) using the `Run` method. Subtests are hierarchic, and one subtest can run another.

3.5 Example tests

We made good progress with our package and tests. It's now time to document our package's usage. We want to do it so that our documentation will never go out of date. Whenever we change our code, our documentation should automatically update itself.

Go has built-in support for what we want and they are called *example tests*. Figure 3.9 shows the `Parse` function's interactive documentation as an example. This documentation will auto-update whenever we update the `Parse` function or the example test.

Figure 3.9 The Parse function's documentation provides an interactive example.


func Parse

```
func Parse(rawURL string) (*URL, error)
```

Parse parses a raw URL into a URL structure.

▼ Example

```
func main() {
    u, err := url.Parse("https://go.dev/play")
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(u)
}
```

Output:

```
https://go.dev/play
```

An example test provides the example code under the "Example" heading and users can click on the Run button to interact with it. Example tests not only help generate runnable examples for users to interactively try our code right inside the documentation, like test functions, but they also verify our code. Let's get into how to write an example test.

3.5.1 Writing an example test

The next listing shows how to write an example test for the `url` package's `Parse` function that will be displayed similarly to Figure 3.9. Example tests are like test functions, but they start with an "`Example`" prefix and don't take a `*T` type. Instead of reporting their result using `*T`, they print it directly to standard output (i.e., using `fmt.Println`, etc.).

Listing 3.11 Writing an example test (url/example_test.go)

```
package url_test


import (
    "fmt"
    "log"
    "github.com/inancgumus/gobyexample/testing/url"    #A
)

func ExampleParse() {                                #B
    u, err := url.Parse("http://go.dev/play")    #C
    if err != nil {                          #C
        log.Fatal(err)                          #C
    }                                     #C
    fmt.Println(u)                          #C
    // Output:                     #D
    // https://go.dev/play            #E
}
```

Let's go over this example test to understand how it works.

First, we create a new file `example_test.go` within the `url` package's directory. Naming the files that house example tests like this is a convention, but it's not required. Then, we put our example test in a new package, `url_test`, for the interactive example to work.

Go usually doesn't let multiple packages within the same directory, but it makes an exception for test packages like `url_test`. The `url` and `url_test` can live in the same directory. The naming pattern `url_test` is a convention and is only available while running tests.

Warning

Test packages should be named after the packages they test. For instance, the `url_test` package tests the `url` package. This is also called black-box testing because the `url_test` package can access only the exported items of the `url` package.

After that, we import the `url` package, as we would for any other package we want to import and use. Since `url` is a different package than the `url_test` package, we can finally write our example test.

We call it `ExampleParse` since we're writing an example test for the `Parse` function. Doing so also enables putting the interactive example next to the `Parse` function within the documentation (as in the figure).

Within the example test, we demonstrate to users how to use the `Parse` function. As shown in the figure, the documentation showcases the code inside the example test as if it were written in a `main` function. The test will fail if it reaches the `log` package's `Fatal` function.

Note

The `Fatal` function causes the example test to end immediately with a status code 1. Since example tests write their results to standard out, the test tool can catch this status code.

Let's now focus on the comment near the end: "// *Output:*". Under the output comment, we write the expected output of our code example. The test will fail if what we output from `Println` doesn't match the expected output. We should keep this comment if we both want to have a high-level test for `Parse` and also provide documentation to users.

3.5.2 Running example tests

Now that we have an example test for the `Parse` function, we can run it like any other test.

```
$ go test -run=ExampleParse$ -v

--- PASS: ExampleParse
```

The `testing` package runs our example test, collects its output, and compares it with the text we wrote under the output comment. Although we don't see its output here, our example test should have output *"https://go.dev/play"* since the example test passed.

Let's assume that in the example test, we mistakenly added something like *"https://google.com"* under the output comment while we passed *"https://go.dev/play"* to `Parse`. In that case, the test would fail, and we would get the following failure message:

```
$ go test -run=ExampleParse$ -v
--- FAIL: ExampleParse (0.00s)
got:
https://go.dev/play
want:
https://google.com
```

Once we upload our module to a public repository, we can see interactive documentation on the Go package server (pkg.go.dev). This website periodically scans information about public modules and packages. See the link for more information: pkg.go.dev/about

Exercise

Write a method that resets the URL fields to zero values and an example test for it.

3.5.3 Wrap up

- Example tests both verify code and provide interactive code examples.
- Example tests have an `Example` prefix and don't take any input.
- The output comment determines an example test's expected output.
- Test packages named after the package they test and enable black-box testing.

Note

Learn more about example tests at the link: blog.golang.org/examples. Also, check out go.dev/doc/comment for more information on documenting Go code.

3.6 Summary

- The `go test` tool runs tests and benchmarks.
- The `testing` package's `*T` type provides methods to report test failures, log messages, etc. The `*B` type provides additional methods for benchmarking.
- Subtests are standalone test functions run by another test function. When combined with table-driven tests, subtests help empower us to write more resilient tests.
- Example tests provide runnable documentation for our packages.

4 Test Coverage and Performance Optimization

This chapter covers

- Writing example tests to generate runnable documentation.
- Measuring test coverage to see what percentage of code is tested.
- Optimizing for performance using benchmarks and profiling.
- Parallel testing to reduce test run time and detect data race issues.

We introduced testing in Go in the previous chapter and crafted a package called `url`.

In this chapter, we'll measure our code's test coverage to ensure we have enough tests. Then, we'll improve our code's performance by benchmarking and profiling. Lastly, we'll look into running tests in parallel to others and detecting possible data race issues.

We'll continue exploring testing in Go in the following chapters.

Note

Find the source code of this chapter at the link:
https://github.com/inancgumus/gobyexample/tree/main/testing

4.1 Test coverage

As a radar scans for objects within a radius, test coverage scans our code to ensure which parts are tested and which are not. Let's delve into test coverage, which is a way of cross-checking to see which parts of code our tests are verifying or, in other words, covering.

Say some of our team members noticed the tests for the `Parse` function aren't enough and are worried about possible bugs. For instance, they wonder what would happen if we wanted to parse URLs with some missing parts, such as *"https://"* or *"foo.com"*. It feels like the tests may not cover every aspect of `Parse` yet. Could we find these issues before they do?

4.1.1 Measuring test coverage

Let's run the tests with the `coverprofile` flag to measure the overall test coverage:

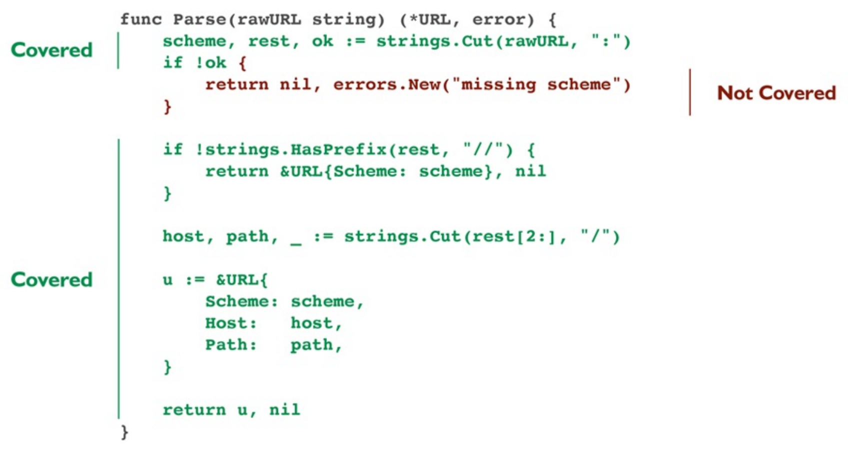
```
$ go test -coverprofile cover.out    #A

PASS
coverage: 87.5% of statements
```

The tool analyzed the `url` package's code and saved a coverage profile to the "`cover.out`" file. It looks like our tests have pretty good coverage: 87.5%. Let's find out what they don't cover by feeding the coverage profile into the coverage tool:

```
$ go tool cover -html=cover.out
```

Running this command will open a browser window and show us the coverage report, which might look similar to Figure 4.1. This figure is best viewed in colors, but the book will be printed in gray color. It might be a good idea to run the command on a local machine.

Figure 4.1 Tests cover every line but the part returning the missing scheme error. Remember that this output is normally colored.



```
func Parse(rawURL string) (*URL, error) {
    scheme, rest, ok := strings.Cut(rawURL, ":")
    if !ok {
        return nil, errors.New("missing scheme")
    }

    if !strings.HasPrefix(rest, "//") {
        return &URL{Scheme: scheme}, nil
    }

    host, path, _ := strings.Cut(rest[2:], "/")

    u := &URL{
        Scheme: scheme,
        Host:   host,
        Path:   path,
    }

    return u, nil
}
```

The green lines are the areas in the code where tests cover ("Covered"), while the red lines are where tests don't cover ("Not Covered"). The rest are gray and not important for the test coverage (the first and last lines in the source code). Looking at the report; our tests exercise every line of code in `Parse` but one: the missing scheme error. Let's add more tests.

Avoiding the browser

If we want to avoid the browser and see the function-by-function coverage report from the command line, we can use the `func` flag as follows. Remember, methods are functions, too.

```
$ go tool cover -func=cover.out

url/url.go:13:   Parse           85.7%
url/url.go:30:   String          100.0%
total:           (statements)    87.5%
```

4.1.2 Reaching 100% test coverage

Now that we have discovered we lack enough tests, we'll increase the test coverage by adding new tests. Our current tests only cover the happy path. This new test will test some edge cases where `Parse` must return an error. See the next listing.

Listing 4.1 Writing an example test (url/url_test.go)

```
func TestParseError(t *testing.T) {
    tests := []struct {
        name string
        uri  string
    }{
        {name: "without_scheme", uri: "go.dev"},

        /* we'll add more tests soon */
    }
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            _, err := Parse(tt.uri)
            if err == nil {                   #A
                t.Errorf("Parse(%q)=nil; want an error", tt.uri)
            }
        })
    }
}
```

The test will fail if `Parse` doesn't return an error. We have only one test case, but we'll add more soon. Let's look at the report without generating a profile file.

This runs tests with coverage:

```
$ go test -cover
```

```
PASS
coverage: 100.0% of statements
```

Our tests now cover every line of code in the `url` package. We could stop here and celebrate since we have 100% coverage. But there is more to the story. Let's find out.

4.1.3 100% test coverage != bug-free code

100% test coverage doesn't mean we have bug-free code. Test coverage only shows us which parts of the code are exercised by tests, but it cannot unearth bugs for us. Let's write some tests and discover to see if our code has bugs despite having full test coverage.

Let's find the bug: empty scheme

The `Parse` function might have a bug. Can it handle URLs with an empty scheme name (i.e., "*://go.dev*")? Let's add a test case to see if this bug exists. See the next listing.

Listing 4.2 Adding an empty scheme test case (url/url_test.go)

```
func TestParseError(t *testing.T) {
    tests := []struct { /* omitted */ }
        {name: "without_scheme", uri: "go.dev"},
        {name: "empty_scheme", uri: "://go.dev"},
    }
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            _, err := Parse(tt.uri)
            if err == nil {
                t.Errorf("Parse(%q)=nil; want an error", tt.uri)
            }
        })
    }
}
```

Let's rerun the test.

```
$ go test -cover
--- FAIL: TestParseError/empty_scheme
    Parse("://go.dev")=nil; want an error
coverage: 100.0% of statements
```

Although the test coverage is 100%, we've discovered a bug!

As the next listing shows, we were parsing the scheme separator and returning an error if the separator was missing. We're now also checking if the scheme was empty.

Listing 4.3 Fix for empty schemes (url/url.go)

```
func Parse(rawURL string) (*URL, error) {
    scheme, rest, ok := strings.Cut(rawURL, ":")
    if !ok || scheme == "" {            #A
        return nil, errors.New("missing scheme")
    }

    if !strings.HasPrefix(rest, "//") {
        return &URL{Scheme: scheme}, nil
    }
    host, path, _ := strings.Cut(rest[2:], "/")

    u := &URL{
        Scheme: scheme,
        Host:   host,
        Path:   path,
    }

    return u, nil
}
```

Rerunning the test with this fix should report a success:

```
$ go test -cover
PASS
coverage: 100.0% of statements
```

This bug hunt was straightforward. Let's focus on another part of our code.

Let's find another bug: nil receiver

Recall from Section 4.1 that `*URL` has a `String` method that reassemblies a URL string.

```
func (u *URL) String() string {
    return fmt.Sprintf("%s://%s/%s", u.Scheme, u.Host, u.Path)
}
```

We suspect there's a bug within `String` when it's called on a `nil` `*URL` pointer, so we're going to put it to the test. As the next listing shows, we're transitioning `TestURLString` to a table-driven approach for clarity. Our first test checks `String`'s behavior with a `nil` `*URL`.

Listing 4.4 Testing String with a nil pointer (url/url_test.go)

```
func TestURLString(t *testing.T) {
    tests := []struct {
        name string
        uri  *URL
        want string
    }{
        { name: "nil", uri: nil, want: "" },    #A
        /* we'll add more test cases soon */
    }
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            got := tt.uri.String()           #A
            if got != tt.want {
                t.Errorf("\ngot  %q\nwant %q\nfor  %#v", got, tt.
            }
        })
    }
}
```

When we run this test, it should fail and trigger a panic due to a `nil` pointer dereference, halting the test run prematurely. A subtest failure should not stop the entire test, but a panic does exactly that, ceasing the parent test abruptly without running the remaining subtests.

```
$ go test -run=TestURLString
--- FAIL: TestURLString
--- FAIL: TestURLString/nil
panic: runtime error: invalid memory address or nil pointer deref
```

Now, how does `String` on a `nil` `*URL` cause such chaos? It's because invoking `String` on a `nil` receiver leads to dereferencing a `nil` pointer, particularly when accessing the fields:

```
func (u *URL) String() string {                    #A

    return fmt.Sprintf("%s://%s/%s", u.Scheme, u.Host, u.Path)
}
```

To make `String` more robust and idiomatic, we'll follow the principle of "*making the zero value useful*." We'll adjust `String` to handle a `nil` receiver gracefully by returning an empty string, thus avoiding the crash. See the improved version in the next listing.

Listing 4.5 Fix for a nil pointer (url/url.go)

```
func (u *URL) String() string {
    if u == nil {
        return ""
    }
    return fmt.Sprintf("%s://%s/%s", u.Scheme, u.Host, u.Path)
}
```

After this tweak, our test should run smoothly without any panics.

```
$ go test -run=TestURLString
PASS
```

Let's continue our bug hunt and discover and fix one more bug.

How is it possible to call a method on a nil pointer?

Internally, a method is a function that takes a receiver input as its first:

```
func String(u *URL) string { /* code */ }
```

Calling `uri.String()` is equal to calling `(*URL).String(uri)`. The part `(*URL)` tells the compiler which type's `String` method to call. We call `*URL`'s `String`, passing it `uri`.

Let's find one more bug: empty URL

`String` reassembles a URL string without checking if the `URL` fields are empty. This might lead it to produce an incorrect URL string. The next listing shows testing this behavior.

Listing 4.6 Testing String with an empty URL (url/url_test.go)

```
func TestURLString(t *testing.T) {
    tests := []struct {
        name string
        uri  *URL
        want string
    }{
        { name: "nil",   uri: nil,    want: "" },
        { name: "empty", uri: &URL{}, want: "" },
    }
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            got := tt.uri.String()
            if got != tt.want {
                t.Errorf("\ngot  %q\nwant %q\nfor  %#v", got, tt.
            }
        })
    }
}
```

Running the test will report a failure:

```
$ go test -run='TestURLString/empty' -v
got  ":///"
want ""
for  &url.URL{Scheme:"", Host:"", Path:""}
--- FAIL: TestURLString/empty
```

We got a string with the scheme and path separators instead of an empty URL string. The next shows a fix for this issue. We now check each field and reassemble a string accordingly.

Listing 4.7 Fix for reassembling a URL (url/url.go)

```
func (u *URL) String() string {
    if u == nil {
        return ""
    }
    var s string
    if sc := u.Scheme; sc != "" {        #A
        s += u.Scheme
        s += "://"           #B
    }
    if h := u.Host; h != "" {
        s += h
    }
    if p := u.Path; p != "" {
        s += "/"
        s += p
    }
    return s
}
```

Now that we've fixed the issue, let's rerun the test.

```
$ go test -cover
PASS..coverage: 100.0% of statements
```

Although we have a fix, `String` might be inefficient. But we can't know that without measuring. We'll delve into how to benchmark to improve our code's performance soon.

Exercise

Although we have complete test coverage, `String` still has a bug. It doesn't handle opaque URLs in the form "data:text/plain". Add a test case and fix the code. You might also need to modify `URL` and `Parse`. Add test cases and fix them, too. Lastly, measure the coverage.

Exercise

Add new test cases to test `String` with a mixture of empty `URL` fields. For instance, a test for a URL without a host, another without a path, and so on.

4.1.4 Wrap up

Our `Parse` function and `String` method are in better shape now. We could continue, but doing so would push us further from our goal of learning to test in Go. We've discovered that test coverage is a tool rather than an end goal. Let's wrap up.

- Test coverage helps find untested code but doesn't guarantee correct code.
- The `coverprofile` flag generates a test coverage profile.
- The `cover` flag measures the test coverage while running tests.
- The `go cover` tool can display the test coverage using a coverage profile.

Note

Learn more about the coverage tool at the link: go.dev/blog/cover

4.2 Benchmarking and optimization

Benchmarking in Go is repeatedly rerunning a piece of code to measure its performance.

Similar to test functions, we write benchmark functions inside test files. Unlike test functions, benchmark functions have a `Benchmark` prefix and take a `*testing.B` input. This struct type has nearly every feature of `*T` (i.e., `Fatalf`) and also supports benchmarking.

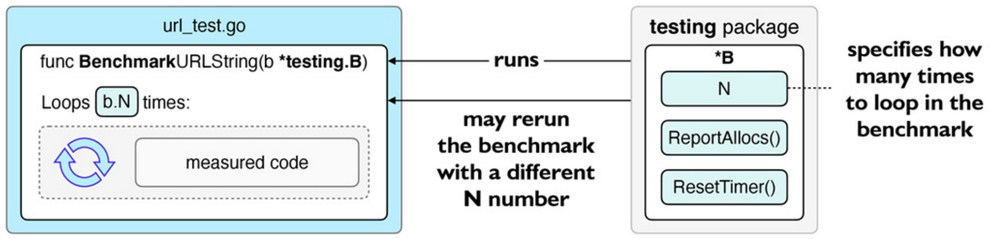
Remember

From now on, we'll say `*B` while referring to `*testing.B` for brevity. Also, sometimes we'll say just benchmarks while talking about benchmark functions.

Say, we want to measure the `String` method's performance. As Figure 4.2 shows, the `testing` package passes a `*B` as `b` when we run our benchmark.

The benchmark will run `String` for `b.N` (iterations count) times in a loop to accurately measure its performance.

Figure 4.2 Benchmarking the code in a loop b.N times to measure performance.


url_test.go
func BenchmarkURLString(b *testing.B)
Loops b.N times:
measured code
runs
may rerun the benchmark with a different N number
testing package
*B
N
ReportAllocs()
ResetTimer()
specifies how many times to loop in the benchmark

The `N` is an integer field of `*B` dynamically adjusted by the `testing` package to achieve a reliable measurement. The `testing` package may rerun the benchmark function many times with gradually increasing `N` numbers until the time elapsed reaches one second (by default).

Avoid optimizing for optimization's sake

Without going into the rabbit hole of benchmarking and optimization, remember that it's better to focus on delivering features and fixing bugs rather than optimizing code unless there is a business need. We should stop if our efforts result in hard-to-maintain code.

4.2.1 Writing and running benchmarks

Now that we're familiar with benchmarking, let's write a benchmark function.

The next listing shows a benchmark function that measures the performance of the `String` method. We name the function with `BenchmarkURLString` to differentiate it from other benchmarks we might add later. After creating a `URL` pointer, we call `String` for `b.N` times.

Listing 4.8 Writing a benchmark for String (url/url_test.go)

```
func BenchmarkURLString(b *testing.B) {

    u := &URL{
        Scheme: "https",
        Host:   "go.dev",
        Path:   "play",
    }
    for range b.N {
        _ = u.String()
    }
}
```

Warning

To get accurate measurements, avoid varying the measured code's behavior depending on `N`. A benchmarked function should not use `N` to change its behavior.

Similar to the `run` flag to run tests, we can run benchmarks with the `bench` flag. We'll also pass the optional `benchmem` flag to get how much memory `String` allocates on average.

```
$ go test -bench=. -benchmem    #A
```

```
BenchmarkURLString-10  11116522  100.7 ns/op  64 B/op  4 allocs/o
```

Here, the `BenchmarkURLString` function ran on 10-CPU cores. The benchmark invoked `String` about 11 million times (11116522), each taking 100.7 nanoseconds on average. Each `String` call requested 64 bytes of heap memory in total by 4 calls on average (4 allocs/op). The "op" means "operation" and denotes a single iteration in the benchmark's loop.

Tip

Test functions are run before benchmarks (we can use the `-v` flag to prove this). We can run benchmarks without running tests using an empty `run` flag: `-bench=. -run=' '`

Setting the benchmark time

We can also change the default benchmark time using the `benchtime` flag.

This adjustment helps us get more accurate results by ensuring that performance is measured over a period that better reflects real-world usage scenarios. Additionally, it might minimize the impact of environmental noise factors (machine load, etc.).

```
$ go test -bench=. -benchmem -benchtime=2s

BenchmarkURLString-10  120111961  99.94 ns/op  64 B/op  4 allocs/
ok      github.com/inancgumus/gobyexample/testing/url      2.610s
```

We set the benchmark for two seconds, but the duration was 2.61 seconds. This is because the benchmark function gets called many times for an accurate result.

Resetting the benchmark timer for accurate measurements

In benchmarking, we sometimes set up resource-intensive tasks like database connections beforehand. Here's how we can do it correctly without skewing the measurements:

```
func BenchmarkMemory(b *testing.B) {
    mem := make([]byte, 1_000_000)        #A
    b.ResetTimer()            #B
    for range b.N {
        filloutBytes(mem)
    }
}
```

Allocating a one-million-byte slice can affect the benchmark results. We reset the timer before the loop to prevent this, ensuring only the relevant code is timed. Comparing results:

```
With ResetTimer   : 20 B/op
Without ResetTimer: 4857 B/op
```

The benchmark inflates the memory allocation stats without resetting, leading to incorrect measurements. For accurate results, we should reset the timer after intensive setup.

4.2.2 Sub-benchmarks

We can use sub-benchmarks to get a richer perspective of `String`'s performance, say, for example, while reassembling URLs with various lengths. Like subtests, the `*B` type has a `Run` method that we can use to run a sub-benchmark. Let's look at the next listing.

Listing 4.9 Writing a benchmark for String (url/url_test.go)

```
func BenchmarkURLStringLong(b *testing.B) {
    for _, n := range []int{1, 10, 100, 1_000} {
        u := &URL{
            Scheme: strings.Repeat("x", n),            #A
            Host:   strings.Repeat("y", n),            #A
            Path:   strings.Repeat("z", n),            #A
        }
        b.Run(fmt.Sprintf("%d", n), func(b *testing.B) {        #
            for range b.N {
                _ = u.String()
            }
        })                           #B
    }
}
```

We're looping over an `int` slice, creating incrementally larger URLs, and running sub-benchmarks to measure `String`'s performance with these URLs.

This setup will give us a clear picture of how the growing size of URLs affects `String`'s performance. We'll use the `bench` flag to run only our new benchmark function this time.

```
$ go test -bench='BenchmarkURLStringLong$' -benchmem    #A

BenchmarkURLStringLong/1-10    12928491   87.75 ns/op     32 B/op
BenchmarkURLStringLong/10-10   12928491   96.47 ns/op    112 B/op
BenchmarkURLStringLong/100-10   7826594   150.8 ns/op    848 B/op
BenchmarkURLStringLong/1000-10  1555650   746.4 ns/op   8192 B/op
```

These four benchmarks prove `String` takes longer to process larger URLs. Still, `String` always makes four allocations (4 allocs/op). Let's investigate where `String` makes these memory allocations so we can see how to optimize it.

Exercise

Interpret the benchmark results. Rerun benchmarks with longer strings.

4.2.3 Profiling: chasing out memory allocations

While benchmarks show us about a piece of code's performance, performance profiling can help us understand why the code acts that way. So we can optimize its performance.

Note

Profiling helps us where to look to improve performance measured by benchmarks.

Our first step is to generate a *memory profile* using the `memprofile` flag as follows. This command will run the benchmarks as before and also generate a memory profile.

```
$ go test -bench='BenchmarkURLStringLong$' -benchmem -memprofile=
```

Now that we have a memory profile file, we can pass this file to the performance profiling tool (pprof) and inspect `String`'s memory allocations using the `list` flag.

```
$ go tool pprof -list String mem.out

27.93GB    27.93GB (flat, cum)   100% of Total
      .          .     29:func (u *URL) String() string {
                                ...
      .          .     33:   var s string
      .          .     34:   if sc := u.Scheme; sc != "" {
      .          .     35:           s += u.Scheme            #A
 3.58GB     3.58GB     36:           s += "://"            #B
      .          .     37:   }
      .          .     38:   if h := u.Host; h != "" {
 6.90GB     6.90GB     39:           s += h            #B
      .          .     40:   }
      .          .     41:   if p := u.Path; p != "" {
 6.88GB     6.88GB     42:           s += "/"            #B
10.57GB    10.57GB     43:           s += p            #B
      .          .     44:   }
      .          .     45:   return s
      .          .     46: }
```

Here, we see that `String` combines strings four times, creating new strings.

Remember that strings are byte slices with a fixed-size underlying byte array. Slicing strings doesn't copy the underlying array. However, combining strings copies the underlying array to a new one (allocates).

Remember

Strings are immutable byte slices that point to an underlying byte array.

Before optimizing `String`, let's save the current results to a file so that we can compare them with the new results later on. Since environmental noise (i.e. system load, hardware temperature, etc.) might affect results, we should repeatedly run benchmarks to minimize the environmental impact. For that, for example, we can use the `count` flag.

```
$ go test -bench='BenchmarkURLStringLong$' -benchmem -count=20 >
```

This command will run the benchmark function twenty times. We can repeat with a larger number if we're unsatisfied with the results. Next, let's optimize the `String` method.

Note

Check out the documentation for more details about diagnosing Go programs: go.dev/doc/diagnostics.

4.2.4 Optimizing the code

Instead of producing more work for the garbage collector and slowing `String` down by making memory allocation requests from the operating system each time we combine strings, we can use the `strings` package's `Builder` type to combine strings. This type can accumulate bytes to the same internal buffer without making additional memory allocations.

The next listing shows the new `String` code that uses a `Builder` to reassemble the URL.

Listing 4.10 Optimizing the String method (url/url_test.go)

```
func (u *URL) String() string {
    if u == nil {
        return ""
    }

    var s strings.Builder                #A

    lenURL := len(u.Scheme) + len("://") +        #B
        len(u.Host) + len("/") +            #B
        len(u.Path)                #B
    s.Grow(lenURL)                #B

    if sc := u.Scheme; sc != "" {
        s.WriteString(sc)                #C
        s.WriteString("://")            #C
    }
    if h := u.Host; h != "" {
        s.WriteString(h)                #C
    }
    if p := u.Path; p != "" {
        s.WriteByte('/')                #D
        s.WriteString(p)                #C
    }

    return s.String()                #E
}
```

Note

Calling the `len` function is cheap (efficient) as `len` reads the length of a string from the string's pre-calculated `length` field without looping over the string to calculate the length.

Once we declare the `Builder` variable `s`, we use `Grow` to set the internal buffer to a large enough size to hold the final URL string. This will allocate a memory block once. Otherwise, `Builder` would inefficiently allocate many times once its buffer is full while we add strings.

After that, we write strings to the buffer using `WriteString` and `WriteByte` without making additional memory allocations. Lastly, we return the accumulated buffer as a string.

Now that we have a new implementation, let's benchmark it and find out if it's efficient.

Exercise

Analyze the memory profile of the new `String` implementation using pprof.

4.2.5 Comparing benchmarks

Benchmark results should be considered relatively. We can't really know if `String` is inefficient before comparing it to another `String` implementation. We already have the old results in the `old.benchmark` file. Let's also benchmark the new implementation and save.

```
$ go test -bench='BenchmarkURLStringLong$' -benchmem -count=20 >
```

The next step is to use the `benchstat` tool to compare the results in a straightforward way. Unlike other tools, this tool isn't builtin and we must install it ourselves as follows.

```
$ go install golang.org/x/perf/cmd/benchstat@latest
```

Note

See the benchstat's documentation for more information on its usage options: pkg.go.dev/golang.org/x/perf/cmd/benchstat.

We can now compare the old and new results as follows.

```
$ benchstat old.benchmark new.benchmark
            |   old    |    new    | difference |
------------+----------+-----------+------------+
sec/op      |  178.9n  |  54.17n   |  -69.73%   |
B/op        |  397.2   |  139.4    |  -64.91%   |
allocs/op   |  4.000   |  1.000    |  -75.00%   |
```

Note

The demonstrated output is simplified for clarity.

Our new implementation is about 3 times more efficient than the previous version. We'll keep the new implementation! Before concluding this section, let's discuss compiler optimizations.

Alternating measurements for accurate results

Our current approach runs benchmarks consecutively for each version (i.e., `-count=20`):

```
OOOOOOOOOOOOOOOOOOOO (benchmark the old version) > old.benchmark
NNNNNNNNNNNNNNNNNNNN (benchmark the new version) > new.benchmark
```

For simple methods like `String`, running benchmarks consecutively is fine. However, this approach might lead to inaccurate results due to environmental factors between each block of measurements. We should use the following approach if we want more reliable results:

```
ONONONONONONONONONONONONONONONONONONONON
```

This approach measures each version once and repeats doing this twenty times:

```
Benchmark the old version once >> old.benchmark        #A
Benchmark the new version once >> new.benchmark        #A
Repeat running these commands 20 times.
```

By alternating the benchmark runs between versions this way, we can mitigate the impact of environmental noise since we'll be measuring both versions under similar conditions.

A straightforward way of doing this is to compile each version using the `c` and `o` flags. For instance, the following commands produce the executables `old.test` and `new.test`.

```
$ go test -c -o old.test

# Run the following after optimizing the old code
$ go test -c -o new.test
```

Now that we have two executable test binaries, we can benchmark them separately:

```
$ ./old.test -test.bench=. -test.benchmem -test.count=1 >> old.be

$ ./new.test -test.bench=. -test.benchmem -test.count=1 >> new.be
# Do this 20 times and then compare the results using Benchstat.
```

The `go test` tool automatically handles this conversion (i.e., from -test.bench to -bench); however, while running a standalone test binary, as shown above, we must prefix each flag with "`-test.`".

4.2.6 Compiler optimizations

We should be careful when we're benchmarking pure or simple functions that can be optimized away by the compiler. Although rare in practice, these optimizations may skew benchmark results. Let's look at the issue and a possible solution.

Discovering the issue

Consider the following pure function example to understand the issue:

```
func Pow(a, n int) int {
    for range n {
        a *= n
    }
    return a
}
```

This function's code is short, has no side effects, and always returns the same result when called with the same values. We can benchmark this function as follows.

```
func BenchmarkPow1(b *testing.B) {
    for range b.N {
        Pow(5, 1)        #A
    }
}
```

The compiler can remove the `Pow` call from the benchmark since it can calculate the function's result at compile time or because the benchmark doesn't use the `Pow`'s result. Looking at the measurements, we can prove this. We see a suspiciously fast operation time.

```
BenchmarkPow1-10  1000000000  0.3119 ns/op
```

This result is not correct. We would have gotten the same result if we had removed the `Pow` call from the benchmark function. This benchmark measures its own loop's performance!

Remember

Under one nanosecond operation time is a red flag we should watch out for.

Here's our benchmark after the compiler optimizations:

```
func BenchmarkPow(b *testing.B) {
    for range b.N { /* empty: no operation */ }
}
```

The compiler detects the `Pow` call in the original benchmark code doesn't affect the rest of the code's outcome and removes the `Pow` call from the benchmark. First, it inlines `Pow`'s code directly into the benchmark to reduce the cost of calling `Pow`. Then, it detects that `Pow`'s result was never used and uses dead code elimination to remove the inlined code.

Fixing the issue

To get an accurate result, we should prevent the compiler from removing the `Pow` call. One common trick is to use an exported package variable named `Sink` as follows.

```
var Sink int            #A


func BenchmarkPow2(b *testing.B) {
    var sink int            #B
    for range b.N {
        sink = Pow(5, 1)        #C
    }
    Sink = sink            #D
}
```

This technique works since it's difficult for the compiler to track cross-package interactions of a package variable like `Sink`, so it can't easily assume

`Pow` is safe to remove. The compiler should assume that the `Pow`'s result is needed somewhere else in the code.

Warning

A common mistake is to use an unexported package variable (e.g., `var sink int`). We should export the variable to make the compiler's optimization efforts more difficult.

We also use the local variable `sink` to cut down CPU memory access costs associated with a package variable, ensuring a more accurate performance evaluation of `Pow`.

Although this method solves the issue, it pollutes the benchmark code. Another drawback is the compiler is improving and this solution is not guaranteed to work in a future release. More importantly, we should use this solution only when we're suspicious of our measurements. We shouldn't blindly apply this technique to every benchmark.

Note

Avoid complicating the benchmarks unless necessary. Clarity matters more.

A better future

Luckily, we *might soon* have a much better way: the `Loop` method. This method can help us safely benchmark our code without having to deal with compiler optimizations ourselves:

```
func BenchmarkURLString(b *testing.B) {

    for b.Loop() {
        Pow(5, 1) // automatically safe-guards this code
    }
}
```

The benchmark uses `Loop` instead of `N`, also telling the compiler to turn off optimizations.

4.2.7 Wrap up

- Benchmarks measure the performance of a piece of code by running it repeatedly. They have a `Benchmark` prefix and take a `*testing.B` input.
- Profiling helps us understand where our code is spending time and memory.
- Benchmarking trivial functions can be inaccurate due to compiler optimizations.

Note

We haven't covered it, but Go also supports fuzz testing. With fuzzing, we can automatically test our code against random inputs to find unhandled areas to improve stability and security. See the link for more information: go.dev/doc/security/fuzz

4.3 Parallel testing

Running tests in parallel helps reduce the total test run time, which is especially useful when we have many tests. Because the `testing` package runs tests sequentially by default, we must mark each that we want to run parallel to others using the `Parallel` method like this:

```
func TestPar1(t *testing.T) {
    t.Parallel()
    time.Sleep(5 * time.Second) // to simulate a long-running tes
    // ...
}
```

Let's add another parallel test and a sequential test.

```
func TestPar2(t *testing.T) {
    t.Parallel()
    time.Sleep(5 * time.Second)
    // ...
}
func TestSeq(t *testing.T) {
    // ...
}
```

We have two parallel tests (`TestPar1` and `TestPar2`) and one sequential test (`TestSeq`). Running these tests will first run all sequential tests and then parallel tests:

```
$ go test -v
=== RUN   TestPar1
=== PAUSE TestPar1        #A
=== RUN   TestPar2
=== PAUSE TestPar2        #A
=== RUN   TestSeq            #B
--- PASS: TestSeq (0.00s)
=== CONT  TestPar1        #C
=== CONT  TestPar2        #C
--- PASS: TestPar2 (5.00s)
--- PASS: TestPar1 (5.00s)
PASS
ok 5.451s
```

The parallel tests paused until the sequential test finished running and ran afterward. Notice that each parallel test takes five seconds to complete. Since they run in parallel, the total test run took only about six seconds instead of ten.

We might want to run tests with reduced parallelism, perhaps because we might run them in a resource-intensive environment. We can control the parallelism using the "`-parallel`" flag:

```
$ go test -v -parallel 1

=== RUN   TestPar1
=== PAUSE TestPar1
=== RUN   TestPar2
=== PAUSE TestPar2
=== RUN   TestSeq
--- PASS: TestSeq (0.00s)
=== CONT  TestPar1
=== CONT  TestPar2
--- PASS: TestPar1 (5.00s)
--- PASS: TestPar2 (5.00s)
PASS
ok 11.065s
```

Each test ran sequentially, doubling the total run time to eleven seconds. This is the default `testing` package behavior when we don't mark any tests with the `Parallel` method. By default, the maximum level of parallelism is set to the number of CPU cores available on the machine, or the number allocated to a virtual machine, effectively the logical CPU count.

Remember

We can set the parallel flag to any number. However, setting it to a number larger than the CPU cores can backfire and increase the test run time instead of reducing it.

Running subtests in parallel

Remember, subtests are separate test functions. To run them in parallel, we must mark both the subtests and their parent test as parallel. If we don't, the parent test runs sequentially, making any parallel tests wait until the parent test completes.

The following tests use the `Parallel` method correctly:

```
func TestPar3(t *testing.T) {
    t.Parallel()
    t.Run("subpar1", func(t *testing.T) {
        t.Parallel()
        time.Sleep(5 * time.Second)
        // ...
    })
    t.Run("subpar2", func(t *testing.T) {
        t.Parallel()
        time.Sleep(5 * time.Second)
        // ...
    })
}
```

Running the tests outputs the following:

```
$ go test -v
...
--- PASS: TestSeq
--- PASS: TestPar2 (5.00s)
--- PASS: TestPar1 (5.00s)
--- PASS: TestPar3 (0.00s)
    --- PASS: TestPar3/subpar2 (5.00s)
    --- PASS: TestPar3/subpar1 (5.00s)
PASS
ok      github.com/inancgumus/gobyexample/testing/parallel 5.492s
```

Once the sequential test was over, four tests ran in parallel (having ten cores on my machine allowed each test to run in parallel). Each took five seconds to complete. Since they ran parallel, the test run took about six seconds instead of twenty seconds.

Detecting data races

Running tests in parallel might also help detect data race issues within our code or tests early on. A data race occurs when multiple goroutines access the same variable simultaneously, and at least one goroutine modifies it. Let's look at an example. Say we have two parallel tests calling `incr`, which increments `counter`:

```
// race_test.go
...
var counter int

func incr() { counter++ }    // line 3

func TestDataRacy(t *testing.T) {
    t.Parallel()
    t.Run("", func(t *testing.T) {
        t.Parallel()
        incr() // ...
    })
    t.Run("", func(t *testing.T) {
        t.Parallel()
        incr() // ...
    })
}
```

Recall that different goroutines run tests. Because `incr` increments a package-level variable `counter` and the tests are running in parallel, this will cause a data race issue. We can catch a potential data race by enabling Go's race detector using the "`-race`" flag:

```
$ go test -run=TestDataRacy -race
```

```
==================
WARNING: DATA RACE
Read at 0x0001044893e8 by goroutine 1:        #A
  incr()
      race_test.go:3 +0x38

Previous write at 0x0001044893e8 by goroutine 2:    #B
  incr()
      race_test.go:3 +0x50
==================
```

We see two subtest goroutines: goroutine 1 and 2. They concurrently read and write to the same `counter` variable (its memory address is 0x0001044893e8). While goroutine 1 was reading `counter`, goroutine 2 was writing the same variable, hence the data race issue:

```
counter++ is the same as counter = counter + 1

                            ^          ^
                          write      read
```

However, the race detector doesn't always run this flawlessly and might not catch every data race. We can improve the hit ratio by setting the count flag to an arbitrarily high number to repeat running the tests as follows:

```
$ go test -run=TestDataRacy -race -count=10
```

Running the test ten times might enable the race detector to spot any data race issues. Still, we should keep in mind that the detector only works on the executed code paths. So, increasing the test coverage to exercise more code paths might help detect more data races.

A final note: the race flag works with `go build` command (and also with `go run`), too. However, since the detector adds assembly code to the final binary to monitor memory access, it can significantly slow down our programs and use more memory. Therefore, we should use it only in test builds or testing environments, not production deployments.

4.4 Summary

- Test coverage helps find untested code but doesn't guarantee correct code.

- Benchmarks measure a code's performance, while profiling gives insights into where the code spends time and memory.
- Running tests in parallel helps reduce the test run time.
- The race detector helps catch data race issues.

5 Structuring Command-Line Programs

This chapter covers

- Crafting idiomatic and user-friendly command-line programs.
- Parsing command-line arguments and flags.
- Extending the `flag` package's functionality by satisfying interfaces.
- Retrieving and using positional arguments for mandatory arguments.
- Validating command-line arguments and flags.

The command line is handy for efficiently performing and automating tasks such as analyzing logs or launching a server. They empower us to increase productivity significantly.

Note

Recall from the last chapter that flags allow users to customize the behavior of a program. For instance, the "`-v`" flag in "go test -v" instructs the tool to print verbose output.

Go has versatile built-in support for crafting cross-platform command-line programs. We can write a program once without additional modifications and deploy it to major platforms, such as Linux or Windows. Moreover, Go's rich standard library makes it straightforward to parse and validate command-line flags and automatically generate program usage information.

This chapter is about "HIT," a program that provides a command-line interface (CLI).

We'll navigate through the `os` and `flag` packages for flag parsing and validation, learning how to extend `flag`. By chapter's end, you'll have a solid foundation for developing cross-platform, user-friendly command-line programs in Go.

Note

Find the source code of this chapter at the link: https://github.com/inancgumus/gobyexample/tree/main/hit/cmd/hit

5.1 Laying out the groundwork

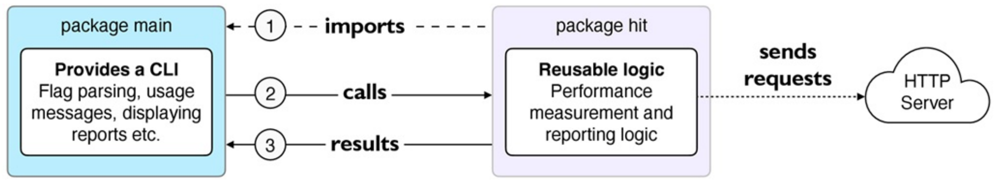
Our program will be a CLI for another package that does the heavy lifting (see Figure 5.1).

Note

We use "CLI", "program", and "tool" interchangeably. They refer to the same thing. Wikipedia defines a CLI as "A command-line interface (CLI) is a means of interacting with a computer program by inputting lines of text called command lines."

We'll have two packages: `main` and `hit`. While we'll develop `main` in this chapter to provide a CLI for users to interact, `hit` will be the goal of the next chapter. The `main` package will parse flags and handle user interactions, while `hit` will measure HTTP performance.

Figure 5.1 Separating the CLI logic from the business logic for reusability. For instance, we could add a REST API in the future that will reuse the business logic.



We'll separate executable commands from importable packages into distinct directories to improve our project's maintainability. Let's look at the following directory structure.

```
.                           -> package hit: Performance measurement

└── cmd                     -> Contains executable commands
    └── hit                 -> package main: HIT CLI tool
    |   └── hit.go          -> func main: Program's entry point
    |   └── hit_test.go    -> Tests for the tool
    └── hitd               -> package main: REST API server (packa
```

We have a `cmd` directory with subdirectories like `hit`, which contains our program, and `hitd`, which could house a future HTTP server. Both can reuse the logic provided by the `hit` package. This separation clarifies the project's organization and boosts reusability.

Note

Remember that while our approach is handy for when a module has `main` and other packages, it's overkill if there is only a single package. Visit the link to learn more about structuring a module: https://go.dev/doc/modules/layout.

First steps for crafting a user-friendly CLI tool

We'll start by printing a usage message with the logo and the flags listed below:

- *-url:* HTTP server URL to send requests.
- *-n*: Number of requests to send to the server.
- *-c:* Concurrency level while sending requests.
- *-rps*: Requests per second.

These flags will allow users to customize the program's behavior. Since our current goal is to show the usage message, we'll parse the flags later. As the next listing shows, we're declaring the tool's logo and usage message in constants and printing them in `main`.

Listing 5.1 Printing the logo and usage message (cmd/hit/hit.go)

```
package main


import (
    "fmt"
    "os"
)

const logo = `                    #A
```

```
 __  __     __     ______
/\ \_\ \   /\ \   /\__  _\
\ \  __ \  \ \ \  \/_/\ \/
 \ \_\ \_\  \ \_\    \ \_\
  \/_/\/_/   \/_/     \/_/`            #A

const usage = `                    #A
Usage:
  -url
       HTTP server URL (required)
  -n
       Number of requests
  -c
       Concurrency level
  -rps
       Requests per second`    #A

func main() {
    fmt.Printf("%s\n%s", logo, usage)

    /* TODO: integrate package hit */
}
```

Let's now build the program using the `go build` tool, passing it an `o` flag to set the executable's name. The following command will produce an executable binary named `hit`.

```
$ go build -o hit ./cmd/hit
```

We can now run this binary in the current directory as follows:

```
$ ./hit                    #A
 __  __     __     ______
/\ \_\ \   /\ \   /\__  _\
\ \  __ \  \ \ \  \/_/\ \/
 \ \_\ \_\  \ \_\    \ \_\
  \/_/\/_/   \/_/     \/_/

Usage:
  -url
       HTTP server URL (required)
  -n
       Number of requests
  -c
       Concurrency level
  -rps
```

```
        Requests per second
```

While `go build` is useful before deployment, `go run` is more practical for rapid development. Instead of rebuilding our program each time we modify its code, we can use the latter:

```
$ go run ./cmd/hit
...the output is the same as above and omitted for brevity...
```

Although our program works, it's not yet easy to maintain. We'll have to change the usage message whenever we modify the flags, which might lead to inconsistent usage messages when flags change. Soon, we'll look at how to automate it using the `flag` package.

Note

Like the `hit` package we'll develop in the next chapter, our tool's name is also `hit`.

5.2 Parsing command-line flags

Now that we have our program's initial version, our next step is to parse flags using:

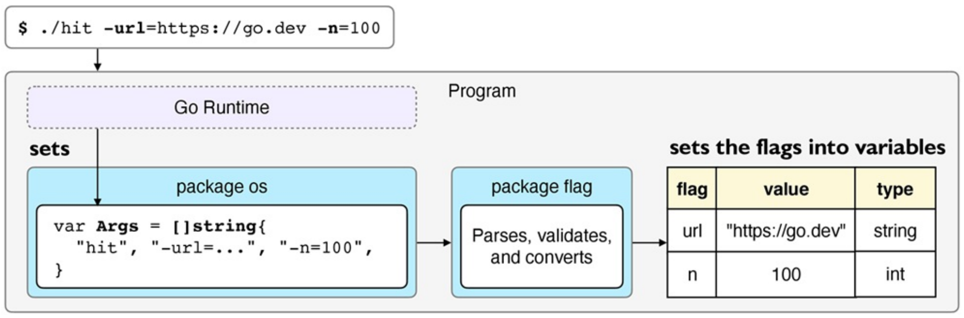
1. The `os` package: Provides access to the command-line arguments.
2. The `flag` package: Handles flag parsing and usage message generation.

Let's look at how we can use them together to parse flags. Figure 5.2 demonstrates running the hit program with the flags url and n (assuming it can handle flags).

Note

n in "-n=100" is the flag's name and 100 is the flag's value.

Figure 5.2 The `flag` package parses and saves the command-line arguments set in `os.Args` by the Go runtime into type-safe regular Go variables we provide.


$ ./hit -url=https://go.dev -n=100
Program
Go Runtime
sets
package os
var Args = []string{
"hit", "-url=...", "-n=100",
}
package flag
Parses, validates, and converts
sets the flags into variables
flag | value | type
url | "https://go.dev" | string
n | 100 | int

The `os` package's `Args` variable (a string slice) contains raw command-line arguments.

But `os` doesn't provide a parser. The `flag` package can parse arguments and set flag values into variables. For example, it can parse "-url=https://go.dev" and set "https://go.dev" into a `string` variable. Or, parse "-n=100" and set 100 into an `int` variable.

Our first goal is to write a custom parser from scratch until switching to the `flag` package. This will familiarize us with command-line arguments, flags, higher-order functions, error wrapping, maps, and more.

Exercise

Write a program that prints `os.Args` to the console. Run the program with different arguments to see what it prints. Compile the program using `go build` and run it again.

5.2.1 Crafting a custom parser

We'll build a custom parser to convert arguments from `os.Args` into variables, setting the stage for adopting the `flag` package later. We'll also learn about higher-order functions, maps, error chains, and the `error` type. But first, let's declare `config` (see the next listing). This will help us store the flag values later in the code once we parse the flags.

Listing 5.2 Implementing the config type (cmd/hit/env.go)

```
package main
```

```
type config struct {
    url  string // url to send requests
    n    int    // n is the number of requests
    c    int    // c is the concurrency level
    rps  int    // rps is the number of requests
}
```

We put the `config` type in a new file called env.go (short for "environment") to manage everything related to the program's external needs, like flags, environment variables, etc. Now that we have a place to store flag values, let's return back to the custom parser.

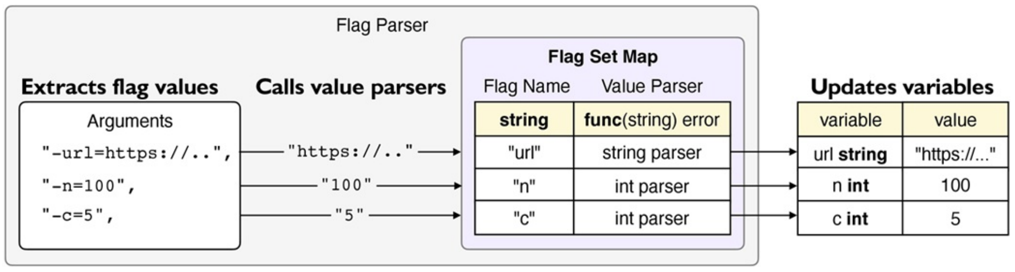
An overview of the custom parser

The custom parser is a function. It takes a `config` to save flags into and a string slice for the command-line arguments that we will inject from `os.Args`.

```
func parseArgs(c *config, args []string) error
```

Let's now look at Figure 5.3 for an overview of the parser's design. The parser extracts flag values from command-line arguments, parses each, and assigns them to variables.

Figure 5.3 The parser maps the flag names to value parsers. These parse flag values they receive from the parser, convert, and save them into variables.



Since flag values are always strings, but variables can have different types, we have a specific value parser function for each variable type we have. Each parser takes a flag value, converts and assigns it to a variable. They return an error if they can't parse the value:

```
func(string) error // <- a value parser function's signature
```

We map flag names to value parsers in a map to quickly fetch a parser by flag name:

```
map[string]func(string) error

    ^        ^
flag name  flag value parser function
```

For instance, we map the url flag to a `string` parser and the n flag to an `int` parser.

Value parsers have the same function signature because map values should be the same type (map keys, too). These parsers only take a flag value, but the target variable they set is not in their signature. But how do they know which variables to update?

Note

Maps are defined together with their key and value types. For instance, `map[string]int` and `map[int]string` are distinct map types. Also, maps are implicit pointers (i.e., functions can modify maps they take) to a data structure in Go Runtime. Lastly, map keys should be comparable types, like strings, but not like function values. Map values can be of any type.

Dive deep: Heap allocation and garbage collection

The `parseArgs` function takes a `config` pointer instead of returning a pointer. This way, we might avoid a potential heap allocation and reduce pressure on the garbage collector. Pointers usually cause an allocation when returned but might not when passed to functions.

However, we shouldn't change a function's API only for optimization. The function takes a pointer also because of future code changes. For instance, an environment variable parser could fill out a `config`, and then we can pass it to `parseArgs`. This way, we can chain many parsers. We'll also set default flag values within a `config` and pass it to `parseArgs`.

You can find more information about reducing the garbage collector pressure and heap allocations, escape analysis, and more at the link:

https://go.dev/doc/gc-guide

Higher-order value parsers

Value parsers are closures and get the variable's pointer from a higher-order function:

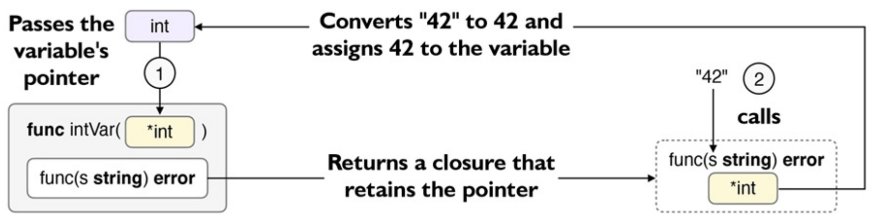
```
func intVar(p *int) func(string) error { // <- higher-order funct

    return func(s string) error {        // <- flag value parser
        var err error
        *p, err = strconv.Atoi(s)
        return err
    }
}
```

Note

A closure returned by a function can retain the function's variables including inputs. Functions that return or take other functions are called higher-order functions.

Let's take a look at Figure 5.4 to understand how `intVar` works. The `intVar` function returns a closure that carries a variable's pointer. Calling the closure converts the flag's `string` value to an `int` value and updates the variable's value through the pointer.

Figure 5.4 Calling `intVar` returns a closure that retains the variable's pointer. Calling the closure can update the original variable's value through the pointer.



Passes the variable's pointer
int
1
func intVar(*int)
func(s string) error
Converts "42" to 42 and assigns 42 to the variable
Returns a closure that retains the pointer
"42"
2
calls
func(s string) error
*int

For instance, say we want to parse the n flag's value and store in the following variable:

```
var n int
```

We pass the `n`'s pointer to `intVar` to get a "`string` to `int` parser" function:

```
parse := intVar(&n)        #A
```

Calling `parse` with "42" indirectly assigns 42 to the `n` variable through its pointer:

```
_ = parse("42")            #A

fmt.Println(n) // prints 42    #B
```

The next listing shows the implementation of `stringVar` and `intVar`. Each is a higher-order function for binding a flag to a variable. The new `parseFunc` type is for clarity.

Listing 5.3 Implementing the flag value parsers (cmd/hit/env.go)

```
type parseFunc func(string) error        #A


func stringVar(p *string) parseFunc {    #B
    return func(s string) error {        #C
        *p = s               #D
        return nil
    }
}

func intVar(p *int) parseFunc {        #E
    return func(s string) error {
        var err error
        *p, err = strconv.Atoi(s)        #F
        return err
    }
}
```

These higher-order functions, like `intVar`, can take different types of variables and return functions with a consistent function signature. This approach enables us to enlist the value parser functions in a map and use them while parsing flags and their values.

Exercise

Implement a `boolVar` parser and use `strconv.ParseBool` to parse string to bool. Implement a `durationVar` parser and use `time.Parse` to parse string to `time.Duration`.

Implementing the custom parser

We're about to start implementing the custom parser. See the next listing.

Listing 5.4 Implementing the flag parser (cmd/hit/env.go)

```
func parseArgs(c *config, args []string) error {

    flagSet := map[string]parseFunc{        #A
        "url": stringVar(&c.url),            #B
        "n":   intVar(&c.n),            #B
        "c":   intVar(&c.c),            #B
        "rps": intVar(&c.rps),            #B
    }
    for _, arg := range args {
        fn, fv, ok := strings.Cut(arg, "=")    #C
        if !ok {
            continue // wrong flag format        #D
        }
        fn = strings.TrimPrefix(fn, "-")        #E
        parseValue, ok := flagSet[fn]        #F
        if !ok {
            continue // not in flagSet        #G
        }
        if err := parseValue(fv); err != nil {    #H
            return fmt.Errorf("invalid value %q for flag %s: %w",
        }
    }
    return nil
}
```

Once we map the flag names and value parsers, we loop over the `args` slice to parse the flag names and values. For each flag, we get a value parser from `flagSet` and use it to parse a flag value. While parsing, we return an error if the value parser can't parse a value.

Passing the "`%w`" verb to `Errorf` wraps the error returned from a value parser into a new `error` value, creating an error chain. We'll see this in action while running the program.

Let's go over this code.

The `flagSet` variable holds a `map` of type `map[string]parseFunc`, which is a key-value pair of flag names and parsers. For instance, let's go over the first element in the map:

```
"url": stringVar(&f.url)
```

This adds an element into the map with the key "url" and a string parser function as the value. Here, `stringVar(&f.url)` returns a value parser that retains the `url` field's pointer.

`flagSet["url"]` returns the string parser function for the url flag, and `true`. It would return `nil` and `false` if we looked up for a non-existent flag name, like: `flagSet["N/A"]`.

Exercise

Add `boolVar` and `durationVar` parsers to the flag set (see the previous exercises).

Tip: Left-align to boost readability and clarity

It's idiomatic to indent fewer times. As in Listing 5.4, we frequently use `continue` or `return` to left-align the code instead of packing the rest in conditionals, like `if`. This makes the code easier to read and understand, helping us see all exit conditions at a glance.

Unlike the right-aligned code below:

```
func parseArgs(c *config, args []string) error {
    flagSet := ...
    for _, arg := range args {
        fn, fv, ok := strings.Cut(arg, "=")
        if ok {
                fn = strings.TrimPrefix(fn, "-")
                parseValue, ok := flagSet[fn]
                if ok {
                        if err := parseValue(fv); err != nil {
                            return fmt.Errorf(...)
                        }
                }
        }
    }
    return nil
}
```

Integration and setting sensible defaults

The next listing shows the parser's integration to `main`. We pass a `config` pointer and the command-line arguments using `Args[1:]`, as we don't need the program name.

Listing 5.5 Integrating the custom parser (cmd/hit/hit.go)

```
func main() {

    var c config
    if err := parseArgs(&c, os.Args[1:]); err != nil {    #A
        fmt.Printf("%s\n%s", err, usage)
        os.Exit(1)
    }
    fmt.Printf(
        "%s\n\nSending %d requests to %q (concurrency: %d)\n",
        logo, c.n, c.url, c.c,
    )
    /* package hit integration here */
}
```

Note

`os.Exit` terminates the program with a numeric status code, ignoring any deferred functions (which we don't have). Status codes are helpful to check a program's success from another program or a script. Usually, a status code 0 means success, while others mean errors.

Let's now try the program with all the flags set.

```
$ go run ./cmd/hit -url="https://go.dev" -n=250 -c=25
```

```
 __  __    __      ______
/\ \_\ \   /\ \   /\__  _\
\ \  __ \  \ \ \  \/_/\ \/
 \ \_\ \_\  \ \_\    \ \_\
  \/_/\/_/   \/_/     \/_/
Sending 250 requests to "https://go.dev" (concurrency: 25)    #A
```

But if we omit some or all the flags, we see zero values.

```
$ go run ./cmd/hit
...
Sending 0 requests to "" (concurrency: 0)
```

Let's fix this problem using sensible defaults. See the next listing.

Listing 5.6 Setting sensible defaults (cmd/hit/hit.go)

```go
func main() {

    c := config{                       #A
        n: 100, // default request count            #A
        c: 1,   // default concurrency level        #A
    }
    if err := parseArgs(&c, os.Args[1:]); err != nil {
        fmt.Printf("%s\n%s", err, usage)
        os.Exit(1)
    }
    fmt.Printf(
        "%s\n\nSending %d requests to %q (concurrency: %d)\n",
        logo, c.n, c.url, c.c,
    )
    /* package hit integration here */
}
```

When n and c flags are omitted, `parseArgs` won't parse them, but the fields will have their default values as we set them before passing the `config`'s pointer. Let's try the program:

```
$ go run ./cmd/hit
...
Sending 100 requests to "" (concurrency: 1)
```

This frees users from having to enter all the flags, making the tool more user-friendly.

Exercise

Use `os.Exit` with different status codes and run the program.

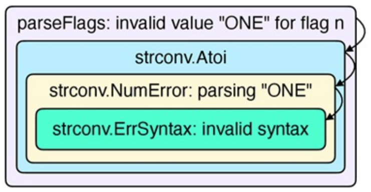
Error chains

Let's also intentionally pass an incorrect flag to see a validation error.

```
$ go run ./cmd/hit -url="https://go.dev" -n=ONE

invalid value "ONE" for flag n: strconv.Atoi: parsing "ONE": inva

Usage of hit:
  -url
       HTTP server URL (required)
  -n
       Number of requests
  -c
       Concurrency level
  -rps
       Requests per second
```

We're observing error unwrapping in action. See Figure 5.5 for an illustration. We first see a parser error, which leads to a flag value parsing error (returns `Atoi`'s error without wrapping it) and then to `Atoi`, followed by a `NumError`, uncovering `ErrSyntax` at the core.

Figure 5.5 Errors can wrap other errors like a Matryoshka doll.


parseFlags: invalid value "ONE" for flag n
strconv.Atoi
strconv.NumError: parsing "ONE"
strconv.ErrSyntax: invalid syntax

This error chain was started when we used `Errorf` and "`%w`" to wrap an `error` in Listing 5.4:

```
func parseArgs(c *config, args []string) error {
    flagSet := map[string]parseFunc{
        "url": stringVar(&c.url),
        "n":   intVar(&c.n),                           #A
        "c":   intVar(&c.c),
        "rps": intVar(&c.rps),
    }
    for _, arg := range args {
        ...
        if err := parseValue(fv); err != nil {
            return fmt.Errorf("invalid value %q for flag %s: %w",
        }
    }
    return nil
}
```

`parseValue` (`intVar`) directly returned an error from `Atoi` without wrapping it (Listing 5.3):

```
func intVar(p *int) parseFunc {
    return func(s string) error {
        var err error
        *p, err = strconv.Atoi(s)    #A
        return err
    }
}
```

That's why we didn't see the error from `parseValue` in the error message. Then, `Atoi` returned a `NumError` pointer wrapping `ErrSyntax`:

```
package strconv

func Atoi(s string) (int, error) {
    ...
    return &NumError{"Atoi", "ONE", ErrSyntax}        #A
    ...
}
```

Since `NumError` (see the sidebar) wraps `ErrSyntax`, we see the "invalid syntax" message:

```
var ErrSyntax = errors.New("invalid syntax")
```

Error wrapping and chains are incredibly helpful for debugging programs. Returning back to our goal, next, we'll rewrite our custom parser from scratch using the `flag` package.

Exercise

Wrap the error returned by `Atoi` using `Errorf` and analyze the result.

Errors are not magic

Errors are values with an `Error` method. Let's look at the built-in `error` interface type:

```
type error interface {

    Error() string
}
```

Calling `Error` returns a string that we can use to print error messages. A type declaring the `Error` method can be passed around as an `error`.

For example, `NumError` is an `error` implementation:

```
type NumError struct {
    Func string
    Num  string
    Err  error                       #A
}

func (e *NumError) Error() string {
    return "strconv." + e.Func + ": " + "parsing " +
        Quote(e.Num) + ": " + e.Err.Error()         #B
}
```

That's why we saw the following message earlier: strconv.Atoi: parsing "ONE". `NumError`'s `Error` method produces and returns a string that combines the wrapped error. We want to create a custom error like `NumError` when we want to provide additional data. Still, rather than creating custom types, wrapping errors using `Errorf` is more practical.

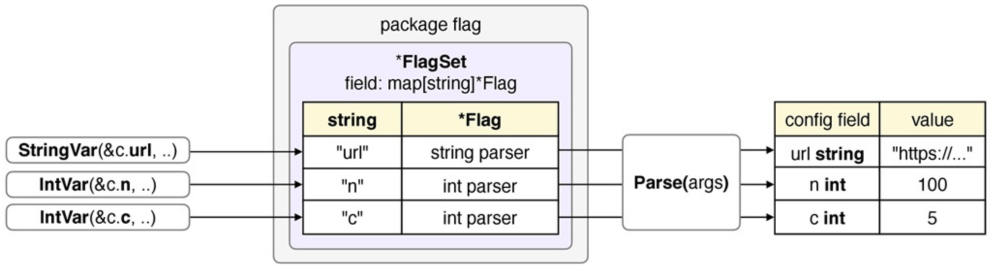
5.2.2 The flag package

Our tool can go a long way with our homemade parser. However, the parser falls short of generating usage messages, silently ignores unsupported flags, and lacks many other features already provided by the `flag` package. First, we'll learn how the `flag` package works. Then, we'll rewrite the `parseArgs` function to use the `flag` package. We'll keep exploring the `flag` package throughout the chapter.

What's a flag set?

Look at Figure 5.6 for an overview of how the `flag` package works. We define our flags on a flag set (`FlagSet`) using methods like `StringVar` and `IntVar`. Then, we call `Parse` to parse the flags we defined and automatically update the variables bound to the flags.

Figure 5.6 `FlagSet` maps flag names to flag value parsers (`Flag`). The `Parse` method parses the flags from command-line arguments and updates the variables. We omit the command-line arguments we pass to `Parse` for brevity.



This is how these methods and types are implemented in Go's `flag` package:

```
package flag
type FlagSet struct {
    formal map[string]*Flag                            #A
    ...
}

func (f *FlagSet) StringVar(p *string, name string, value string,
func (f *FlagSet) IntVar   (p *int,    name string, value int,

func (f *FlagSet) Parse(arguments []string) error
...
type Flag struct {
    Name     string    #C
    Usage    string    #C
    DefValue string    #C
    Value    Value    #D
}
```

More or less similar to our custom parser, a `FlagSet` struct has a map field that maps flag names to flag types: `map[string]*Flag`. Each `Flag` contains a flag's name, default value, usage string, and a specific parser that can parse the flag's value.

We use `StringVar` and `IntVar` to define a `Flag` on a `FlagSet` with a specific value parser, such as a `string` or `int` value parser. These parsers are hidden from our view. Luckily, we can add a `Flag` with a *custom value parser* to a `FlagSet` (which we'll learn how to do soon).

Now, let's take a look at how we can create a flag set and use it to parse flags.

Note

Unlike our previous parser, we can create as many flag sets as we want, each with its own flags. Creating new flag sets is helpful, for instance, when creating subcommands and testing. Each can have a specific flag set and hence flags. For instance, "`commit`" in "`git commit -m 'Fix #1200'`" is a subcommand, as is "`test`" in "`go test -v`". Or "`run`" in "`go run`".

Getting started with flag sets

For example, let's define a new flag set using the `NewFlagSet` function as follows:

```
fs := flag.NewFlagSet("hit", flag.ContinueOnError)    #A
```

The first argument is the flag set's name, typically the program's name. `ContinueOnError` tells the flag set to continue instead of exiting the program with `os.Exit` if parsing fails.

Now that we have a flag set, we can bind a `config` field to a `Flag`. For instance:

```
var c config
fs.StringVar(&c.url, "url", "", "HTTP server URL ")
```

Here, `StringVar` adds a string `Flag` to the `fs` flag set. The arguments are for linking a variable (`c.url`) to a flag (`"url"`), setting a default (empty for now), and explaining its use.

We can now call the `Parse` method to parse the url flag:

```
args := []string{"-url=https://go.dev"}
err := fs.Parse(args)
```

We'll see the `url` field is updated when we print it:

```
fmt.Println(c.url) // prints https://go.dev
```

The flag set can also generate usage messages. Now that we defined a flag, once we integrate this code to `parseArgs`, we can see the usage message like this:

```
$ go run ./cmd/hit -h    #A

Usage of hit:
  -url string
        HTTP server URL
```

We can send the flag's usage more descriptive by wrapping "URL" in backticks:

```
flag.StringVar(&c.url, "url", "", "HTTP server `URL` (required)")
```

This way, the flag set will display URL next to the flag in the usage message:

```
$ go run ./cmd/hit -h
Usage of hit:
  -url URL
        HTTP server URL  (required)
```

Now, it's clearer that the flag expects a URL. We'll explore other interesting usages later.

Note

The `flag` package has a package-level default flag set called `flag.CommandLine` and functions (i.e., `flag.StringVar`) using it. We'll avoid them as the default flag set can automatically exit the program and write to standard error, making testing harder.

Rewriting the parser

Now that we're familiar with the `FlagSet` and `Flag` types, let's update our previous parser code. The next listing shows a rewritten `parseArgs` function.

Listing 5.7 Parsing flags using a flag set (cmd/hit/env.go)

```
package main

import "flag"

type config struct {
    url string // url to send requests
    n   int    // n is the number of requests
    c   int    // c is the concurrency level
    rps int    // rps is the number of requests
}

func parseArgs(c *config, args []string) error {
    fs := flag.NewFlagSet("hit", flag.ContinueOnError)    #A

    fs.StringVar(&c.url, "url", "", "HTTP server URL ")
    fs.IntVar(&c.n, "n", c.n, "Number of requests")
    fs.IntVar(&c.c, "c", c.c, "Concurrency level")
    fs.IntVar(&c.rps, "rps", c.rps, "Requests per second")

    return fs.Parse(args)
}
```

The last two flags now have sensible defaults that they get from `c`. They will default to these values when omitted. We confine the flag set in the `parseArgs` function. This way, if necessary, we can change the `flag` package to a better solution in the future without updating the rest of our code.

We have created a new flag set for the hit command and set it to continue when an error occurs. Then, we defined all our flags—url as a string, c, n, and rps as integers. We've also set their default flag values from the `config` pointer passed to `parseArgs`.

Exercise

Create another flag set in a new program with different flags.

Exercise

Visit the `flag` package's documentation and add a `DurationVar` to the flag set. pkg.go.dev/flag#DurationVar

Dive Deep: Flag definition methods

A flag set has additional methods for defining flags including `BoolVar`, `DurationVar`, `Float64Var`, `Int64Var`, `TextVar`, and so on. The last one is especially useful:

```
func (f *FlagSet) TextVar(

    p      encoding.TextUnmarshaler,
    name   string,
    value  encoding.TextMarshaler,
    usage  string,
)
```

The first parameter requires a type that implements the `TextUnmarshaler` interface, and the third requires one that implements the `TextMarshaler` interface. We can use `TextVar` once one of our types implements those interfaces. For instance, let's look at the `net.IP` type:

```
func (ip IP)  MarshalText() ([]byte, error)    // encoding.TextMa

func (ip *IP) UnmarshalText(text []byte) error // encoding.TextUn
```

Since IP implements these interfaces, we can parse a flag to an `IP` value as follows:

```
var ip net.IP

fs.TextVar(&ip, "ip", &ip, "`IP` address")
fs.Parse([]string{"-ip", "10.0.0.1"})
fmt.Println(ip) // prints 10.0.0.1
```

We can also call the `IP` methods on `ip` (since it's a `net.IP`):

```
fmt.Println(ip.IsPrivate()) // prints true
```

Updating the main function

Now that we have integrated the `flag` package, we also need to update the `main` function. Currently, `main` prints an error and shows the usage message when parsing fails:

```
func main() {
    c := config{
        n: 100, // default request count
        c: 1,   // default concurrency level
    }
    if err := parseArgs(&c, os.Args[1:]); err != nil {
        fmt.Printf("%s\n%s", err, usage)
        os.Exit(1)
    }
    fmt.Printf(
        "%s\n\nSending %d requests to %q (concurrency: %d)\n",
        logo, c.n, c.url, c.c,
    )
    /* package hit integration here */
}
```

However, when the parsing fails, a flag set automatically prints out an error and usage message to the standard error (by default). For instance, let's take a look at the following.

```
$ go run ./cmd/hit -c=ONE                #A

invalid value "ONE" for flag -c: parse error
Usage of hit:                      #A
  -c int                          #A
        Concurrency level (default 1)       #A
  -n int                          #A
        Number of requests (default 100)       #A
  -rps int                       #A
        Requests per second            #A
  -url URL                       #A
        HTTP server URL                 #A
invalid value "ONE" for flag -c: parse error    #A
Usage:                           #B
  -url                           #B
        HTTP server URL (required)           #B
  -n                           #B
        Number of requests            #B
  -c                           #B
        Concurrency level                #B
  -rps                           #B
        Requests per second            #B
```

We're currently outputting duplicate error and usage messages.

The solution is in the next listing. We now skip the message printing while still preserving the `os.Exit` to return a status code. Because the flag set's error output is configured to `ContinueOnError`, it won't yield a status code. Conversely, if we had configured the error output as `ExitOnError`, then the flag set would return a status code of 2 if the parse fails.

Listing 5.8 Removing error and usage messages (cmd/hit/env.go)

```
package main

import (
    "fmt"
    "os"
)

const logo = `
 __  __      __      ______
/\ \_\ \   /\ \   /\__  _\
\ \  __ \  \ \ \  \/_/\ \/
 \ \_\ \_\  \ \_\    \ \_\
  \/_/\/_/   \/_/     \/_/`

// const usage = `...`                    #A

func main() {
    c := config{
        n: 100, // default request count
        c: 1,   // default concurrency level
    }
    if err := parseArgs(&c, os.Args[1:]); err != nil {
        os.Exit(1)
    }
    fmt.Printf(
        "%s\n\nSending %d requests to %q (concurrency: %d)\n",
        logo, c.n, c.url, c.c,
    )
    /* package hit integration here */
}
```

We can also drop the `usage` constant since the flag set auto-generates it. The usage message will always remain consistent with our flags, improving the user experience.

Trying out the new parser

Let's first run the program with the "-h" flag (help) to see the usage message.

```
$ go run ./cmd/hit -h    #A

Usage of hit:
  -c int
        Concurrency level (default 1)
  -n int
        Number of requests (default 100)
  -rps int
        Requests per second
  -url URL
        HTTP server URL
```

We see the defaults for the c and n flags are printed since we previously set them in `main`:

```
c := config{
    n: 100, // default request count
    c: 1,   // default concurrency level
}
```

Let's see if the defaults show up if we run the program without any flags.

```
$ go run ./cmd/hit
Sending 100 requests to "" (concurrency: 1)
```

We see an empty string since we haven't yet defined a default for the url flag.

Note

The usage message is displayed with the "`-h`" or "`-help`" flags or if parsing fails. We can also force a flag set from the code to print a usage message. More on this soon.

Next, let's run the program without the defaults.

```
$ go run ./cmd/hit -c=5 -n=25 -url="https://go.dev"
```

```
Sending 25 requests to "https://go.dev" (concurrency: 5)
```

We can also provide the flags without an equal sign and also with a double-dash:

```
$ go run ./cmd/hit -c 5 -n 25 -url "https://go.dev"
...
$ go run ./cmd/hit --c 5 --n 25 --url "https://go.dev"
...
```

Lastly, let's look at an error message.

```
$ go run ./cmd/hit -n TWO
invalid value "TWO" for flag -n: parse error
Usage of hit:
  -c int
        Concurrency level (default 1)
  -n int
        Number of requests (default 100)
  -rps int
        Requests per second
  -url URL
        HTTP server URL
```

With this, we conclude our getting started tour of flag parsing.

Exercise

Create sub-commands using multiple flag sets with specific flags. Use the second element of `os.Args` (the first is the program name) to determine the flag set requested by a user from the command line. Call that flag set's `Parse` with the rest of the arguments.

5.2.3 Wrap up

Writing a custom parser from scratch helped us learn more about flag handling and some other useful Go features. We then smoothly upgraded our tool to the `flag` package. Our tool is now more capable of flag parsing and displaying user messages.

In the following sections, we'll keep improving the tool. Let's wrap up.

- `os.Args` variable is a string slice of command-line arguments.
- The `flag` package can parse flags, generate usage messages, and more.
- `flag.FlagSet` associates flags with `Flag` values.
- `flag.Flag` contains flag information and updates a variable via a value parser.
- `FlagSet` methods such as `StringVar` and `IntVar` add flags to a `FlagSet`.
- Higher-order functions are those that return or take other functions.
- `os.Exit` exits the program with a status code, bypassing deferred functions.
- `fmt.Errorf` with `"%w"` wraps an `error` in a new error, creating an error chain.

5.3 Custom flag types

Our parser validates the n and c flags as integers but doesn't check if they're positive integers. For instance:

```
$ go run ./cmd/hit -n -1

Sending -1 requests to "" (concurrency: 1)

$ go run ./cmd/hit -c 0
Sending 100 requests to "" (concurrency: 0)
```

We don't currently send HTTP requests, but when we do, negative or zero values for these flags would be meaningless as they go against our ultimate goal. So, we'll implement a custom flag type that can validate whether a flag is a positive integer. This will boost robustness and print more precise error messages for invalid inputs.

5.3.1 Dynamic values

Recall from Section 5.2.2 that a flag set maps flag names to `Flag` pointers.

```
type FlagSet struct {
    map[string]*Flag
    ...
}
```

When we define a flag, it's registered on a flag set's internal map. For instance:

```
fs.StringVar(&c.url, "url", "", "HTTP server `URL`")
```

```
fs.IntVar(&c.n, "n", c.n, "Number of requests")
```

These add two flags to the map, one for a `string` and another for an `int` variable. While the map stores the flags as `Flag` pointers of the same type, what makes it possible to define flags that can dynamically work with different variable types? Let's look at the `Flag` type.

Remember

A `Flag` binds a command-line flag to a variable.

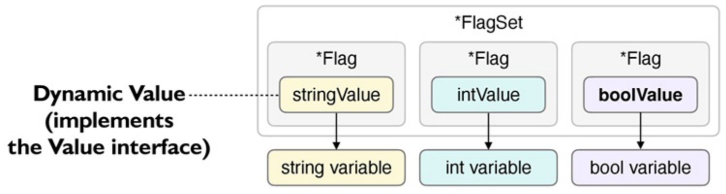
Each `Flag` comes with a *dynamic value* field of type the `Value` interface:

```
type Flag struct {
    Name     string
    Usage    string
    DefValue string
    Value    Value  // <-- flag's dynamic value    #A
}
```

To keep the concept simple, we referred to these dynamic value parts of flags as value parsers. It's time to learn how they work and adjust our terminology.

As Figure 5.7 shows, the dynamic value field enables `Flag` instances to work with various variable types. For instance, calling `IntVar` defines and sets a `Flag`'s dynamic value to an `intValue`, which can convert a flag value and assign it to an `int` variable. Or, calling `StringVar` defines a `Flag` with a dynamic value of a `stringValue` for string variables.

Figure 5.7 Flags can handle many variable types thanks to their dynamic value part.



In summary, `Flag` values have a dynamic field to handle any variable type. This field's type is the `Value` interface, which has many implementations, like `stringValue`, `intValue`, and `boolValue`. We'll also add our own implementation.

Tip

Idiomatic interfaces are lean and centered on shared behavior. For instance, in a FlagSet, flags are represented by concrete Flag instances, not interfaces. The Value field within a Flag is the only interface that reflects its dynamic behavior. The remaining fields—Name, Usage, and DefValue—are concrete, as they do not require the flexibility of an interface.

5.3.2 Value interface

Let's dive into the `Value` interface as it's crucial for developing a custom dynamic value:

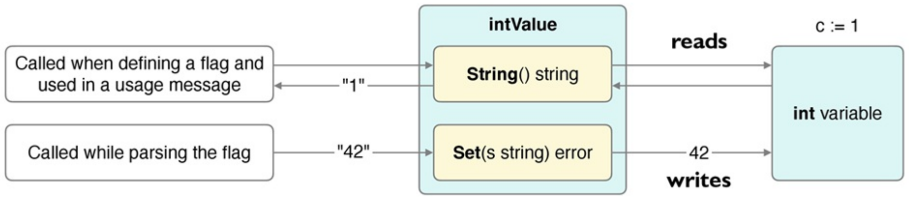
```
// Value is the interface to the dynamic value stored in a flag.
// Set is called once, in command line order, for each flag prese
type Value interface {
    String() string    #A
    Set(string) error    #B
}
```

Figure 5.8 shows `intValue` to demonstrate how a `Value` implementation works.

`Value` has two methods to produce usage messages and manage variables bound:

- `String` returns the variable's value bound to a `Flag`. A flag set calls this method once to set the `Flag`'s default value once we define a flag. This shows up while generating a usage message (e.g., "(default 1)" next to the c flag).
- `Set` parses a flag's value and sets it to a variable (i.e. "42" becomes 42).

Figure 5.8 The `intValue` is a `Value` implementation that can handle `int` variables.


intValue
Called when defining a flag and used in a usage message
"1"
String() string
reads
c := 1
int variable
Called while parsing the flag
"42"
Set(s string) error
42
writes

To wrap up, `String` is called to set a flag's default value after defining a flag on a flag set, and `Set` is called to parse and set a flag's value to a variable bound. Now that we're familiar with how dynamic values work, let's implement a dynamic value type in the next subsection.

Exercise

Implement a type that satisfies the `Value` interface.

5.3.3 Crafting a dynamic value type

For example, we use `IntVar` to define the following flags:

```
fs.IntVar(&c.n, "n", c.n, "Number of requests")
fs.IntVar(&c.c, "c", c.c, "Concurrency level")
fs.IntVar(&c.rps, "rps", c.rps, "Requests per second")
```

`IntVar` adds a `*Flag` with an `intValue` as its dynamic value to the flag set. An `intValue` can validate if the string input is an integer. But we also need to validate if it's a positive integer, so we're creating a `positiveIntValue` to validate positive integers. See the next listing.

Listing 5.9 Implementing a custom dynamic value (cmd/hit/env.go)

```
type positiveIntValue int                         #A


func newPositiveIntValue(p *int) *positiveIntValue {
    return (*positiveIntValue)(p)                 #B
}

func (n *positiveIntValue) String() string {
    return strconv.Itoa(int(*n))                 #C
}

func (n *positiveIntValue) Set(s string) error {
    v, err := strconv.ParseInt(s, 0, strconv.IntSize)    #D
    if err != nil {
        return err
    }
    if v <= 0 {
        return errors.New("should be greater than zero")
    }
    *n = positiveIntValue(v)                 #E

    return nil
}
```

Note

Dynamic values are typically named typeValue, like intValue, durationValue, and so on.

Using `ParseInt`, we turn a `string` into an `int`. Passing 0 for the base lets it figure out the number system, like "1000" and "1_000" for decimal and "0x50" for hex. Setting `IntSize` is for handling big numbers, up to 64 bits. This will allow us to handle almost any integer.

Our dynamic value type satisfies the `Value` interface (with `String` and `Set`) and will allow us to declare a flag that must be a positive integer. While `String` converts and returns an `int` as a `string`, `Set` assigns a string input as an integer to a variable if it's a positive integer.

Remember

Types with similar underlying types can be converted to each other.

Moreover, we can turn any `int` pointer type into a `Value` implementation. We can call `String` and `Set` methods on the latter once we convert an `*int` to a `*positiveIntValue`. For instance, let's declare an `int` and then convert its pointer to a `*positiveIntValue`:

```
n := 1
p := newPositiveIntValue(&n)
```

Here, `p` points to `n`'s memory address.

Since `p` is a `*positiveIntValue`, we can call `String` and `Set` on it:

```
fmt.Println(p.String()) // prints "1"

_ = p.Set("10")         // Sets n to 10
fmt.Println(n)          // prints "10"
```

Calling `String` returns `n`'s value and `Set` sets `n` through the pointer `p`. With `positiveIntValue`, we can turn any `int` pointer to a type that has `String` and `Set` methods. Since this type satisfies the `Value` interface, we can enforce positive integer flags.

Note

Since `*positiveIntValue` is a `Stringer` (implements `String`), we could print `p` without explicitly calling the `String` method: `fmt.Println(p)`.

strconv.ParseInt vs. strconv.Atoi

Both `ParseInt` and `Atoi` functions can convert a string to an integer. Still, `ParseInt` is superior. For example, it can parse `1_000` to `1000` or `0xff` to `255`. See the link for more details: pkg.go.dev/strconv#ParseInt

5.3.4 Defining a flag with a custom dynamic value

It's that moment now. In the next listing, we're redefining the c, n, and rps flags using our new dynamic value type: `positiveIntValue`, which is a `Value` implementation.

Listing 5.10 Defining a custom flag type (cmd/hit/env.go)

```
func parseArgs(c *config, args []string) error {
    fs := flag.NewFlagSet("hit", flag.ContinueOnError)

    fs.StringVar(&c.url, "url", "", "HTTP server `URL` (required)
    fs.Var(newPositiveIntValue(&c.n), "n", "Number of requests")
    fs.Var(newPositiveIntValue(&c.c), "c", "Concurrency level")
    fs.Var(newPositiveIntValue(&c.rps), "rps", "Requests per seco

    return fs.Parse(args)
}
```

This time, instead of using methods like `StringVar`, we use `Var`:

```
func (f *FlagSet) Var(value Value, name string, usage string)
```

`Var` expects a `Value` interface. Since `*positiveIntValue` implements `Value`, we can pass one here to define a flag using it as its dynamic value. Unlike others, `Var` doesn't let us pass a default value. Luckily, the flag set will call our custom type's `String` to do that. This default value will show up in the usage message next to the flag.

Now that we defined custom flags, let's see if they reject non-positive numbers:

```
$ go run ./cmd/hit -n 0

invalid value "0" for flag -n: should be greater than zero
Usage of hit:
  -c value
        Concurrency level (default 1)
  -n value
        Number of requests (default 100)
  -rps value
        Requests per second
  -url URL
        HTTP server URL (required)

$ go run ./cmd/hit -c -1
invalid value "-1" for flag -c: should be greater than zero
...
```

Let's also check if we can still validate non-integer flags:

```
$ go run ./cmd/hit -rps gopher

invalid value "gopher" for flag -rps: strconv.ParseInt: parsing "
...
```

Thanks to `ParseInt`, we can also pass integers in other formats:

```
$ go run ./cmd/hit -url https://go.dev -n 10_000 -c 0xff

Sending 10000 requests to "https://go.dev" (concurrency: 255)
```

Isn't it great that we can implicitly implement an interface declared in another package (`flag.Value`)? Because our type says nothing about the `Value` interface, it's decoupled from the interface. We can even use it in other parts of our code when needed.

Exercise

Implement a custom flag type that can validate positive `Duration` values. Add a new flag called timeout to the `parseArgs`'s flag set. Add a custom flag type that accepts only the following: GET, POST, and PUT.

Dive deep: The usage of Var is plenty

Like `Flag`, `Var` is at the heart of the `FlagSet` type's design. For instance, other flag set methods use `Var` to define a `Flag` with a dynamic value:

```
func (f *FlagSet) StringVar(p *string, name string, value string,
    f.Var(newStringValue(value, p), name, usage)
}
```

`StringVar` defines a `Flag` with a `stringValue` similar to what we've done:

```
func newStringValue(val string, p *string) *stringValue {
    *p = val            #A
    return (*stringValue)(p)
}
```

Other flag definition functions follow the same approach.

5.3.5 Wrap up

- A `Flag` has a dynamic value part to handle different variable types.
- `Value` has two methods: `String` and `Set`.
- Custom dynamic value types implement `Value`.
- `Var` defines a `Flag` on a `FlagSet` with a dynamic value type.

5.4 Positional arguments

Flags are optional arguments to customize our tool's behavior, but the URL is a must. The tool can't work without it for HTTP requests, so we'll make the URL a *positional argument*. This way, it'll be crystal clear that users must provide a URL. Since the URL will no longer be a flag, we'll need to retrieve it from the command-line arguments ourselves. We'll also need to customize the usage message, as the flag set can't generate one without a flag defined.

5.4.1 Flags vs. positional arguments

Let's start by understanding the differences between flags and positional arguments.

Say we have the following flag set (the same one we used so far):

```
fs := flag.NewFlagSet("hit", flag.ContinueOnError)
fs.StringVar(&c.url, "url", ...)
fs.Var(newPositiveIntValue(&c.n), "n", ...)
```

Currently, the url flag can be in any position:

```
fs.Parse([]string{"-url=http://go.dev", "-n=10"})    #A

fs.Parse([]string{"-n=10", "-url=http://go.dev"})
```

A *positional argument*, on the other hand, is sensitive to its position (hence the name):

```
fs.Parse([]string{"-n=10", "http://go.dev"})
```

Notice that the last argument isn't a flag. It's just "http://go.dev" without "-url=". We can access this first positional argument using the `FlagSet`'s `Arg` method:

```
<
```

We can retrieve the URL using `Arg(0)` even if there are fewer or no flags before it:

```
fs.Parse([]string{"http://go.dev"})

fmt.Println(fs.Arg(0)) // prints "http://go.dev"
```

Since flag parsing stops with the first non-flag argument, a positional one should always come after flags. For instance, the following is incorrect and the flag won't be parsed:

```
fs.Parse([]string{"http://go.dev", "-n=1"})

fmt.Println(fs.Arg(0)) // prints "https://go.dev"
fmt.Println(fs.Arg(1)) // prints "-n=1"
```

Here, both arguments become positional, with the last one no longer treated as a flag. To wrap up, we can call `Arg` to retrieve a positional argument after calling `Parse`.

Remember

Like flags, positional arguments are only available after calling `Parse`.

Exercise

Write a program that prints all the positional arguments. Use the `Args` method at the link to determine the number of positional arguments: pkg.go.dev/flag#Args

5.4.2 Customizing usage messages

Say we have the following flag set (the same one above):

```
fs := flag.NewFlagSet("hit", flag.ContinueOnError)
fs.Var(newPositiveIntValue(&c.n), "n", ...)
```

The usage message will no longer include the url flag as we don't define the url flag. Luckily, it's straightforward to generate a custom usage message with the `Usage` field. This field's type is a function neither takes nor returns a value:

```
type FlagSet struct {
    Usage func()        #A
    ...
}
```

Note

`Usage` is called when an error occurs or help is requested (e.g., with the "-h" flag).

For instance, we can assign this function to `Usage` to define a custom usage message:

```
fs.Usage = func() {                               #A
    fmt.Fprintf(fs.Output(), "usage: %s [options] url\n", fs.Name
    fs.PrintDefaults()                               #C
}
```

Let's go over the previous code to understand it better.

- `fs.Usage = func() {...}` overwrites the flag set's usage message behavior.
- `fs.Name` returns the flag set's name: "hit."
- `fs.Output` returns the flag set's output target (typically standard error).
- `fmt.Fprintf(fs.Output(), ...)` prints a usage header to the flag set's output.
- `fs.PrintDefaults()` lists all the flags with their descriptions and defaults.

While printing the usage header, we use `Name` instead of directly using "hit" for consistency. This way, we can keep aligned with the flag set's name if we change it later.

Similarly, calling `Fprintf` with `Output` is also for consistency purposes. Otherwise, it would be confusing for users. For instance, say, the flag set prints to standard error, but `Fprintf` prints to standard out. A script running our tool would have to track both outputs. Determining which one contains errors or the usual output would be confusing.

Calling `Usage` would print the following to the standard error (by default):

```
usage: hit [options] url            #A
  -n value                #B
      Number of requests (default 100)    #B
```

In summary, we can customize a flag set's usage message with `Usage`. Passing `Output` to `Fprintf` keeps where we write consistent. Using `Name` returns a flag set's name.

Exercise

Set the `Usage` function to another function that prints a different usage message.

5.4.3 Setting a positional argument

Now that we looked at how to work with positional arguments and generate custom usage messages, let's modify our parser to switch the url flag to a positional argument. In the following listing, we set a custom usage message using `Usage`, remove the url flag definition, parse the flags, and then get the URL from the first positional argument using `Arg(0)`.

Listing 5.11 Switching to a positional argument (cmd/hit/env.go)

```
func parseArgs(c *config, args []string) error {
    fs := flag.NewFlagSet("hit", flag.ContinueOnError)
    fs.Usage = func() {
        fmt.Fprintf(fs.Output(), "usage: %s [options] url\n", fs.
        fs.PrintDefaults()
    }

    fs.Var(newPositiveIntValue(&c.n), "n", "Number of requests")
    fs.Var(newPositiveIntValue(&c.c), "c", "Concurrency level")
    fs.Var(newPositiveIntValue(&c.rps), "rps", "Requests per seco
    if err := fs.Parse(args); err != nil {
        return err
    }
    c.url = fs.Arg(0)                                      #B

    return nil
}
```

The final usage message looks like the following:

```
$ go run ./cmd/hit -h
usage: hit [options] url
  -c value
```

```
        Concurrency level (default 1)
  -n value
        Number of requests (default 100)
  -rps value
        Requests per second
```

Now, let's run the command with flags and a positional argument together:

```
$ go run ./cmd/hit -c 2 -n 100 http://go.dev
Sending 100 requests to "http://go.dev" (concurrency: 2)
```

We've supplied the positional argument last and observed it in the output.

Exercise

Call `Arg` before calling `Parse` and observe what changes.

5.4.4 Wrap up

Our tool is getting better at each section. Switching the URL to a positional argument suggests users always provide it, separating it from optional parameters. Let's wrap up.

- Defined flags and positional arguments are available only after parsing ends.
- `Arg` returns positional arguments provided after flags (if any).
- Setting the `Usage` function allows customizing the flag set's usage message.
- `Name` returns the flag set's name.
- `Output` returns the flag set's output (standard error by default).
- Passing `Output` to `Fprintf` allows us to print where the flag set prints.
- `PrintDefaults` prints the usage messages of the flags defined.

5.5 Post-parse flag validation

The parser validates flags, but it misses some edge cases. For instance, it doesn't catch when the required argument URL isn't provided, nor does it check if the concurrency level is less than the number of requests. We'll now focus on reporting these edge cases.

This will simplify troubleshooting for our users and make the tool more user-friendly. Since a flag set doesn't support validating mandatory arguments or interdependent flags, we'll validate them by checking the `config` fields after parsing ends ourselves.

5.5.1 Writing a custom validator

Listing 5.12 demonstrates a new function that validates the `config` fields. We'll call this function from the parser soon. Inside the function, each validation step returns an error using the `argError` function to produce similar errors to the `flag` package.

For instance, it'll produce the following error if the n flag is less than the c flag.

```
$ go run ./cmd/hit -n 1 -c 2 "https://go.dev"
invalid value "1" for flag -n: should be greater than -c: "2"
```

This way, the tool's error texts remain consistent with others and won't surprise users.

Listing 5.12 Implementing a custom validator (cmd/hit/env.go)

```
func validateArgs(c *config) error {

    const urlArg = "url argument"

    u, err := url.Parse(c.url)
    if err != nil {
        return argError(c.url, urlArg, err)
    }
    if c.url == "" || u.Host == "" || u.Scheme == "" {
        return argError(c.url, urlArg, errors.New("requires a val
    }
    if c.n < c.c {
        err := fmt.Errorf(`should be greater than -c: "%d"`, c.c)
        return argError(c.n, "flag -n", err)
    }

    return nil
}
```

```
// argError returns an error message for an invalid argument.
func argError(value any, arg string, err error) error {
    return fmt.Errorf(`invalid value "%v" for %s: %w`, value, arg
}
```

The `validateArgs` function is written top-to-bottom and is easy to read. Writing code top-to-bottom reflects the execution order in runtime, making the logic flow more explicit. Keeping validation in one function streamlines reading, upkeep, and grasp.

Yet, we shouldn't take this as a hard rule against breaking up a function if it gets too complex. It's tough to set a strict rule for when to do this, but generally, begin with a solid, single-piece code and split it up as it grows complex and demands more reuse.

Note

`any` represents an empty interface: `interface{}`. Since this interface has no methods, it doesn't put any restrictions, and every type can satisfy it. This way, `argError` can take any type, like int or string: `argError(42, ..)` or `argError("http://dev", ..)`

5.5.2 Integrating the custom validator

It's time to integrate the custom validator function we implemented. See the next listing. We call `validateArgs` after parsing to validate the fields. Since the flag set isn't involved in this validation, we force the flag set to print the usage message if an error occurs.

Listing 5.13 Integrating the custom validator (cmd/hit/env.go)

```
func parseArgs(c *config, args []string) error {
    fs := flag.NewFlagSet("hit", flag.ContinueOnError)
    fs.Usage = func() {
        fmt.Fprintf(fs.Output(), "usage: %s [options] url\n", fs.
        fs.PrintDefaults()
    }

    fs.Var(newPositiveIntValue(&c.n), "n", "Number of requests")
    fs.Var(newPositiveIntValue(&c.c), "c", "Concurrency level")
    fs.Var(newPositiveIntValue(&c.rps), "rps", "Requests per seco
    if err := fs.Parse(args); err != nil {
        return err
    }
    c.url = fs.Arg(0)

    if err := validateArgs(c); err != nil {    #A
        fmt.Fprintln(fs.Output(), err)       #B
        fs.Usage()                #C
        return err
    }

    return nil
}
```

Let's try out the code. We should see validation errors.

```
$ go run ./cmd/hit
invalid value "" for argument url: requires a valid url

$ go run ./cmd/hit -n 1 -c 2 "https://go.dev"
invalid value "1" for flag -n: should be greater than -c: "2"
```

These errors will simplify troubleshooting for users when they pass incorrect arguments.

5.6 Exercises

The following exercises involve defining a flag set, validating flags, and generating a usage message. Of course, you're not expected to implement the core logic—for instance, you're not expected to implement git from scratch!

1. Write a CLI for one of the git subcommands. You can see them by typing: `git -h`
2. Write a CLI for the go test subcommand. Find the flags by: `go help testflag`
3. Write a CLI for one of your favorite or common commands, perhaps `ls`.

5.7 Summary

- `os.Args` string slice variable holds command-line arguments.
- The `flag` package handles flag parsing and usage message generation.

- `FlagSet` ties flags to variables, with a `*Flag` for each.
- `FlagSet.Parse` parses flags and updates variables.
- `Flag` binds a variable to a flag, supporting multiple types via a dynamic value.
- Custom flags implement the `Value` interface, definable with `FlagSet.Var`.
- `FlagSet.Arg` retrieves a positional argument.
- `FlagSet` lacks support for mandatory or interdependent flags.

6 Testable Command-Line Programs

This chapter covers

- Testing command-line programs.
- Isolating dependencies using interfaces to craft testable code.
- Glimpsing into the io.Reader and io.Writer interfaces.

Previously, we crafted an idiomatic and user-friendly command-line program, worked on manually parsing command-line arguments, and explored and extended the flag package. We've wrapped the tool but skipped automated testing and manually tested it to keep things straightforward. It's time to write idiomatic and automated tests for the HIT tool.

Note

Find the source code of this chapter at the link: https://github.com/inancgumus/gobyexample/tree/main/hit/cmd/hit

6.1 Testing executable programs

Many people in the wild use interesting and unnecessarily complicated techniques to test CLI tools. Testing a CLI tool should be no di

fferent from testing any other code. Still, the `main` function's black-box nature —it takes no inputs or outputs—does make it tricky.

We'll start by explaining why `main` is cumbersome to test and find out flexible ways to test it. Then, we'll move on to writing high-level tests to verify our program acts as expected. Lastly, we'll zoom in on the parser with detailed tests and ensure the edge cases are covered.

Note

While `main` is like a black box, it still has side effects since it typically uses globals, like standard output. One can run the program, capture its inputs and outputs, and observe it from outside. However, this would require more code and won't help write testable code.

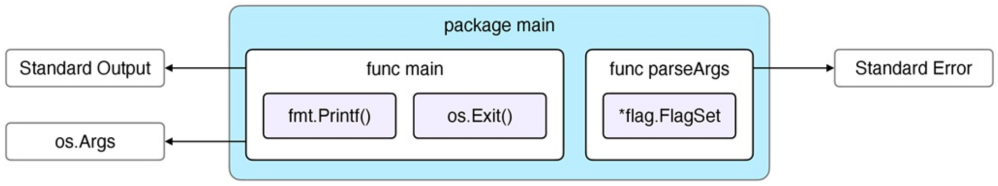
6.1.1 Why is the main package cumbersome to test?

The short answer is `main` uses globals that are out of our *direct* control:

- Reads command-line arguments from the global variable `os.Args`.
- Prints directly to standard output with `Printf`.
- Calls `parseArgs` that prints to standard error with `Fprintf` and `FlagSet`.

Figure 6.1 illustrates this list in a diagram.

Figure 6.1 Uncontrolled external dependencies of our program.



This list includes examples from our program, and it's just a starting point—there's always more to consider with different programs. For reliable testing, isolation, and control are crucial; otherwise, we get hit with flaky tests. Or, other issues that are challenging to debug.

Note

A flaky test is a test that randomly becomes successful or fails on every other run.

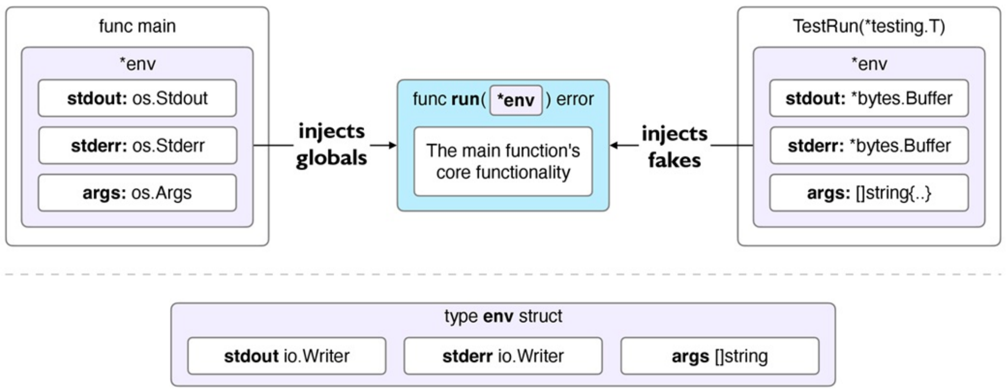
Moreover, parallel testing the code that uses globals can be tricky. We would need to handle concurrent access using mutexes or orchestrate interactions through channels, or we might have to avoid parallel testing altogether, which could lead to longer test times.

To wrap up, these testability issues aren't specific to the `main` function—any code that leans on globals, hides side effects, or is difficult to observe poses a challenge to testing.

6.1.2 Designing testable executable programs

Making `main` testable may seem daunting since it does not accept input parameters nor return values. But there's an effective way out: refactoring `main` to delegate its responsibilities to a testable function called `run`. See Figure 6.2 for an illustration.

Figure 6.2 Moving `main`'s code into a `run` function that accepts an `*env` struct. This allows `main` to inject real globals to `run` while tests can inject controlled fakes.



This refactoring won't affect the program's behavior. But it'll change the entire scenery when we inject fake inputs and outputs from tests into the `run` function.

We extract the core functionality of `main` into a new function, `run`, which takes an `*env` struct. This struct serves as a vessel for dependencies, letting us inject reals or fakes as needed. This way, we can transform the "black box" code of `main` into a testable unit.

We'll first create the `env` struct type and learn about the `io.Writer` interface. Then, we'll move on to the `main` function and extract its functionality to the `run` function. Lastly, we'll update the parser code to adapt to the new changes. By the end, we'll have a testable code.

Abstracting dependencies

To support both real and fake values, `env` has `io.Writer` interface fields to *decouple* our code from standard output and error streams `os.Stdout` and `os.Stderr`.

See the next listing.

Listing 6.1 Declaring the env type (cmd/hit/env.go)

```
type env struct {

    stdout io.Writer // stdout abstracts standard output
    stderr io.Writer // stderr abstracts standard error
    args   []string  // args are command-line arguments
    dry    bool      // dry enables dry mode        #A
}
```

The `Writer` interface is extremely simple and can be satisfied with this `Write` method:

```
type Writer interface {

    Write(p []byte) (n int, err error)    #A
}
```

This interface is one of the most ubiquitous interfaces in the standard library and standardizes and abstracts writing to an output stream, like a file or network connection. Think of it like `OutputStream` in Java, `Stream.Write` in C#, or `RawIOBase.write` in Python.

As shown in Figure 6.2 earlier, we can assign `os.Stdout` (an `*os.File` variable) or a `*bytes.Buffer` to an `env`'s `stdout` field (or `stderr`) because they satisfy `io.Writer`. Not only that, we can even assign a network connection to each and make our program serve over the network (using `net.Conn`). The possibilities are endless with the `Writer` interface.

Dive deep: io.Writer and io.Reader

The `Writer`'s contract allows writing a byte slice efficiently to an output stream in chunks:

```
// Write writes len(p) bytes from p to the underlying data stream
// It returns the number of bytes written from p (0 <= n <= len(p
// and any error encountered that caused the write to stop early.
//
// Write must return a non-nil error if it returns n < len(p).
// Write must not modify the slice data, even temporarily.
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

To handle this efficiently, `Write` expects a byte slice, moving the responsibility of allocating a byte slice to the caller. If `Write` were to return a slice, a `Writer` would be forced to allocate a new slice for every `Write` call. This would be a waste especially when dealing with large data.

Similarly, `io.Reader` facilitates the efficient reading of byte slices from an input stream:

```
// Read reads up to len(p) bytes into p from the underlying data
// It returns the number of bytes read (0 <= n <= len(p)), and an
// If n < len(p), Read must return a non-nil error explaining why
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

Just as Write hands off the duty of byte slice allocation to the caller, Read requires a pre-allocated byte slice to fill with data. This avoids unnecessary allocations on each call, optimizing memory usage. If `Read` were to allocate a new slice, it would lead to inefficiencies similar to those we avoid with `Write`, especially in resource-intensive applications.

Now that we looked at their design, let's combine both to write a simple echo program:

```
func main() {
    io.Copy(os.Stdout, os.Stdin)
}
```

Running this program will echo what we input to the console:

```
$ go run .
hello  <- reads our input from os.Stdin
hello  <- echoes this to os.Stdout
```

The `Copy` function takes a `Writer` and a `Reader`. Here, it reads from `Stdin` (as a `Reader`) and writes to `Stdout` (as a `Writer`). `Copy` holds an internal byte slice to copy data efficiently. Using this program we can transfer gigabytes of with minimal memory overhead (32 KB):

```
$ go run . < from.mov > to.mov # copies from.mov to to.mov
```

This is just the tip of the iceberg. We can do many interesting things with these interfaces.

Decoupling

Now that we have the `env` type to abstract external dependencies, we can move on to refactoring the `main` function. Here's the current one to compare the differences later:

```
func main() {
    c := config{
        n: 100, // default request count
        c: 1,   // default concurrency level
    }
    if err := parseArgs(&c, os.Args[1:]); err != nil {
        os.Exit(1)                        #A
    }
    fmt.Printf(                            #B
        "%s\n\nSending %d requests to %q (concurrency: %d)\n",
        logo, c.n, c.url, c.c,
    )
    /* TODO: integrate package hit */
}
```

In the next listing, we extract `main`'s body and move it to `run`. Then, we call `run` with global dependencies. Unlike `main`, `run` returns an `error` instead of calling `os.Exit`. Otherwise, tests may end abruptly without being able to observe errors. We also pass standard output streams to `Fprintf` and `parseArgs` to capture and observe their output from tests.

Listing 6.2 Calling run from main (cmd/hit/hit.go)

```
func main() {
    e := &env{                            #A
```

```
        stdout: os.Stdout,                        #A
        stderr: os.Stderr,                        #A
        args:   os.Args,                            #A
    }                                   #A
    if err := run(e); err != nil {
        os.Exit(1)                            #B
    }
}

func run(e *env) error {                            #C
    c := config{
        n: 100, // default request count
        c: 1,   // default concurrency level
    }
    if err := parseArgs(&c, e.args[1:], e.stderr); err != nil {
        return err                            #E
    }

    fmt.Fprintf(                                #F
        e.stdout,                                #F
        "%s\n\nSending %d requests to %q (concurrency: %d)\n",
        logo, c.n, c.url, c.c,
    )

    return nil
}
```

The `run` function is a testable replica of the earlier `main`. The `main`'s earlier core functionality is now inside `run` and no longer depends on globals. When `main` calls `run`, it works with globals. When called by tests, `run` can work with fakes. The `main` function itself has become extremely simple, and there is no longer a need to test it. Instead, we can test `run`.

While we're passing an entire `env` struct to the `run` function, we don't do so for `parseArgs` and only passing `stderr`. The difference is that `run` executes the entire program and needs `env`'s every field, while `parseArgs` only needs to write to standard error.

Tip

Avoid functions taking more than what they need to improve maintainability and clarity.

Also, taking a `Writer` as input instead of getting it in an `env` makes `parseArgs` easier to understand and test. If it were to take a `Writer` in an `env`, we would need to create an `env` in each test, put a `Writer` in it, and pass it to `parseArgs`. This would make tests harder to read and maintain. Still, we could make it take an `env` if it needed other fields in `env`. We should weigh and decide for each specific situation, as there is always a trade-off.

Decoupling the parser's output

The `run` function is ready to test, but we'll also adjust `parseArgs` since it has to take a `Writer`. In the next listing, we add a `Writer` input and use it to set the flag set's output using `SetOutput` (which takes a `Writer`). This way, tests can observe the error and usage messages. Out of tests, `parseArgs` outputs to `os.Stderr`, as `main` supplies an `env` with `os.Stderr` to `run`.

Listing 6.3 Setting the flag set's output (cmd/hit/env.go)

```
func parseArgs(c *config, args []string, stderr io.Writer) error

    fs := flag.NewFlagSet("hit", flag.ContinueOnError)
    fs.SetOutput(stderr)                                #B

    fs.Usage = func() {
        fmt.Fprintf(fs.Output(), "usage: %s [options] url\n", fs.
        fs.PrintDefaults()                            #C
    }

    fs.Var(newPositiveIntValue(&c.n), "n", "Number of requests")
    fs.Var(newPositiveIntValue(&c.c), "c", "Concurrency level")
    fs.Var(newPositiveIntValue(&c.rps), "rps", "Requests per seco
    if err := fs.Parse(args); err != nil {
        return err
    }
    c.url = fs.Arg(0)

    if err := validateArgs(c); err != nil {
        fmt.Fprintln(fs.Output(), err)                    #C
        fs.Usage()                              #C
        return err
    }

    return nil
}
```

Earlier, we designed `parseArgs` to write to the same output as the flag set. So, we didn't need to make many changes. We were done once we set the flag set's output to `stderr`. Taking a `Writer` as an input makes `parseArgs` flexible because we can pass in any `Writer`. While tests can pass a "fake" that captures the output, `main` can pass in `os.Stderr`.

Preparing for the hit package

Before wrapping up, let's prepare for the upcoming `hit` package. When the `dry` field we added earlier to `env` is enabled, `run` won't use the `hit` package. See the next listing.

Listing 6.4 Adding runHit and dry (cmd/hit/hit.go)

```
func run(e *env) error {
    c := config{
        n: 100, // default request count
        c: 1,   // default concurrency level
    }
    if err := parseArgs(&c, e.args[1:], e.stderr); err != nil {
        return err
    }
    fmt.Fprintf(
        e.stdout,
        "%s\n\nSending %d requests to %q (concurrency: %d)\n",
        logo, c.n, c.url, c.c,
    )
    if e.dry {               #A
        return nil           #A
    }                   #A

    return runHit(e, &c)            #B
}

func runHit(e *env, c *config) error {    #B
    /* TODO: integrate the hit package */
    return nil               #B
}                    #B
```

This conditional makes sure that tests solely focus on testing our tool rather than the `hit` package's internals. We don't always have to use interfaces for testing if we can do it with a simple conditional. An interface would be needed if we wanted to simulate `hit`'s interactions (or if we needed a switch between different tools) to test how our code responds to them.

We're now prepared for the `hit` package and will integrate it in the next chapter.

Wrap up

Before getting into testing, let's manually test to see if our program still works.

```
$ go run ./cmd/hit "https://go.dev"

 __  __      __      ______
/\ \_\ \   /\ \   /\__  _\
\ \  __ \  \ \ \  \/_/\ \/
 \ \_\ \_\  \ \_\    \ \_\
  \/_/\/_/   \/_/     \/_/
Sending 100 requests to "https://go.dev" (concurrency: 1)

$ go run ./cmd/hit -n "HIT"
invalid value "HIT" for flag -n: strconv.ParseInt: parsing "HIT":
usage: hit [options] url
  -c value
        Concurrency level (default 1)
  -n value
        Number of requests (default 100)
  -rps value
        Requests per second
```

The program's behavior hasn't changed, but it's freed from global dependencies.

This subsection has glimpsed the power of Go in injecting dependencies. By using interfaces like `Writer`, we can avoid the complexity of external dependency-injection frameworks, keeping our code lean and testing straightforward. This simplicity enables us to easily transform our program into a testable one, ensuring it's as dependable as simple.

6.1.3 High-level tests

We'll start with high-level tests. Since diagnosing errors from these is like finding a needle in a haystack, we'll keep their number to a minimum. We'll handle the remaining edge cases from the parser's unit tests. Pinpointing issues in unit tests is typically more straightforward.

Streamlining tests with a test environment

Before getting started, let's remember the `env` type.

```
type env struct {
    stdout io.Writer // stdout abstracts standard output
    stderr io.Writer // stderr abstracts standard error
    args   []string  // args are command-line arguments
    dry    bool      // dry enables dry mode
}
```

Because `stdout` and `stderr` are a `Writer`, we can't read from them directly (or easily). Instead, we can assign a `*bytes.Buffer` to each field and then read from it. `*Buffer` is a `Writer` that accumulates what it's written to it into its internal byte slice. For instance:

```
var buf bytes.Buffer            #A
buf.Write([]byte("hello world"))        #B
buf.String() // returns "hello world"    #C
```

Now that we learned about `Buffer`, let's assign a `*Buffer` to `stdout` and `stderr`:

```
var bout bytes.Buffer
var berr bytes.Buffer

err := run(&env{
    stdout: &bout,    #A
    stderr: &berr,    #A
    args  : []string{"hit", "https://go.dev"},
    dry   : true,
})
```

Note

We assign a `*Buffer` (i.e., `&bout`) since `Buffer` doesn't satisfy `Writer`. Only the `*Buffer`'s method set has a `Write` method since its `Write` method is declared with a pointer receiver.

The program will write to the buffers, and then tests can read from them:

```
bout.String() // returns what's written to stdout
berr.String() // returns what's written to stderr
```

However, doing this setup for each of our tests can be a hassle and clutter the tests even though we'll have a limited number of them. We'll introduce a new type to streamline testing. Similar to what we've done with the `env` type, we'll craft `testEnv` to streamline observing standard outputs from tests. It wraps `env` and has additional `Buffer` fields.

See the next listing.

Listing 6.5 Declaring the testEnv type (cmd/hit/hit_test.go)

```
package main

import "bytes"

type testEnv struct {
    env    env
    stdout bytes.Buffer               #A
    stderr bytes.Buffer               #A
}
```

Dive deep: Make zero values useful

We declare `stdout` and `stderr` as a `Buffer`, not as a pointer: `*Buffer`. This way, Go can automatically initialize them to zero `Buffer` values that we can use right away. Otherwise, we would have to initialize each ourselves and face possible nil-pointer issues.

For instance, let's declare a `testEnv`, write something, and print the buffer. The following is possible because the fields are initialized with a usable `Buffer` immediately.

```
var t testEnv
t.stdout.WriteString("hello")        #A
```

```
_ = t.stdout.String()             #B
```

If we had used a `*Buffer` for the fields, we would face a `nil` pointer dereference unless we explicitly initialized the fields. See the example below.

```
type testEnv {
    env    env
    stdout *bytes.Buffer
    stderr *bytes.Buffer
}

...

// panics:
var t testEnv
t.stdout.WriteString("hello")     #A

...

// ok:
t = testEnv{
    stdout: &bytes.Buffer{},
    stderr: &bytes.Buffer{},
}
t.stdout.WriteString("hello")    #B
```

We should use zero values where possible. They reduce initialization overhead and potential errors, making our code cleaner and more idiomatic.

Of course, we shouldn't take this too far.

For instance, putting a `sync.Mutex` (a classical mutex) in a type with value receiver methods is incorrect. A `sync.Mutex` shouldn't be copied, but value receiver methods would copy it:

```
type value struct {
    mu sync.Mutex
    // other fields
}

func (v value) do() {
    v.mu.Lock()
    defer v.mu.Unlock()
    // do something
}
```

Whenever the `value`'s do method is called, the `mu` field will be copied:

```
var v value
v.do() // mu is copied
v.do() // mu is copied
```

Instead, we can either define a pointer `Mutex` field or use pointer receiver methods:

```
func (v *value) do() { ... }
```

Here, we use a pointer receiver method to avoid copying the mutex. The linters are also smart about this and warn us whenever this copying occurs, even before compiling our code.

Streamlining tests with a function

See the next listing for the last step before testing. The `testRun` function prepares a ready-to-use test environment with sane defaults. Like a non-test run, the `args` field gets the tool's name. The function also initializes a `testEnv` and sets the `stdout` and `stderr` fields. Lastly, it triggers a dry run and returns `testEnv` to observe the `run`'s output.

Listing 6.6 Declaring the testRun function (cmd/hit/hit_test.go)

```
func testRun(args ...string) (*testEnv, error) {    #A

    var t testEnv                    #B
    t.env = env{                    #C
        args:   append([]string{"hit"}, args...),     #D
        stdout: &t.stdout,
        stderr: &t.stderr,
        dry:    true,
    }
    return &t, run(&t.env)
}
```

`testRun` is a convenience function that helps tests to run the program and observe its output. Now that we have everything in place, let's get started with testing.

Dive deep: Variadic functions

`testRun` is variadic because it takes variadic string values: `args ...string`. The `args` parameter turns into a `[]string` within the function. This makes it easier to use `testRun` without declaring a string slice beforehand. Still, we can pass it a slice if we want.

```
testRun("-c=2", "-n=1", "invalid-url")
testRun([]string{"-c=2", "-n=1", "invalid-url"}...)        #A
```

Passing arguments without a slice creates a new slice that points to a new backing array that contains those elements. However, passing a slice passes a slice with the same backing array of the slice: the function can change the elements. If we want to avoid this, we shouldn't pass a slice, or before calling the function, pass a copy of the slice:

```
clone := append([]string{}, original...}
testRun(clone)
```

We're creating an empty string slice (`[]string{}`) and then appending the original slice's elements to that slice (`originalSlice...`). The resulting slice will have a new backing array that contains the same elements of the original slice. The testRun function can no longer modify the original slice's elements since the cloned slice has a separate backing array.

Writing high-level tests

Now that we have `testEnv` to provide a test environment and `testRun` to execute the `run` function and observe the output, let's write high-level tests in the next listing. While the first test tests the tool's behavior with a valid input, the second test does the opposite. We observe the standard output and error streams to see what's being written to them.

Listing 6.7 High-level tests (cmd/hit/hit_test.go)

```
func TestRunValidInput(t *testing.T) {

    t.Parallel()
    e, err := testRun("http://go.dev")
    if err != nil {
        t.Fatalf("got %q;\nwant nil err", err)
    }
    if n := e.stdout.Len(); n == 0 {
        t.Errorf("stdout = 0 bytes; want >0")
    }
    if n, out := e.stderr.Len(), e.stderr.String(); n != 0 {
        t.Errorf("stderr = %d bytes; want 0; stderr:\n%s", n, out
    }
}

func TestRunInvalidInput(t *testing.T) {
    t.Parallel()

    e, err := testRun("-c=2", "-n=1", "invalid-url")    #A
    if err == nil {
        t.Fatalf("got nil; want err")
    }
    if n := e.stderr.Len(); n == 0 {
        t.Error("stderr = 0 bytes; want >0")
    }
}
```

Note

`Len` returns the number of bytes written. `String` returns what's written as a string.

We could add more tests, but we intentionally avoided doing so as errors from high-level tests can be challenging to understand and diagnose. Usually, it is more complicated to guess the root cause of these errors. Let's give these tests a try.

```
$ go test -v
--- PASS: TestRunValidInput (0.00s)
--- PASS: TestRunInvalidInput (0.00s)
```

We've tested our program within an isolated and test-controlled environment.

Exercise

Intentionally break argument handling in the parser code and observe test errors.

6.1.4 Testing the parser

Now that we've tested the tool with high-level tests, it's time to go deeper and test the parser code. Failures from these kinds of granular tests (i.e., unit tests) are typically easier to figure out and act upon than high-level tests. Still, we'll intentionally ignore covering every edge case to keep the test code short and leave those as an exercise for you.

As the next listing shows, we declare the `parseArgsTest` type to reuse it from happy and sad path tests. We should avoid complicating test tables and tests and avoid putting every test within a single test function. Separating the tests like we did will simplify their code.

Listing 6.8 Adding a shared test case type (cmd/hit/env_test.go)

```
package main


import "testing"

type parseArgsTest struct {
    name string
    args []string
    want config
}
```

Now that we have a test case type, we can implement the parser tests. Before starting, let's remember the `parseArgs` function's signature. The tests are in the next listing.

Listing 6.9 Testing with valid flags (cmd/hit/env_test.go)

```
func TestParseArgsValidInput(t *testing.T) {

    t.Parallel()                    #A

    for _, tt := range []parseArgsTest{
        {
            name: "all_flags",
            args: []string{"-n=10", "-c=5", "-rps=5", "http://tes
            want: config{n: 10, c: 5, rps: 5, url: "http://test"}
        },

        // exercise: test with a mixture of flags
    } {
        t.Run(tt.name, func(t *testing.T) {
            t.Parallel()                #A

            var got config
            if err := parseArgs(&got, tt.args, io.Discard); err !
                t.Fatalf("parseArgs() error = %v, want no error",
            }
            if got != tt.want {
                t.Errorf("flags = %+v, want %+v", got, tt.want)
            }
        })
    }
}

func TestParseArgsInvalidInput(t *testing.T) {
    t.Parallel()

    for _, tt := range []parseArgsTest{
        {name: "n_syntax",   args: []string{"-n=ONE", "http://tes
        {name: "n_zero",     args: []string{"-n=0",   "http://tes
        {name: "n_negative", args: []string{"-n=-1",  "http://tes

        // exercise: test other error conditions
    } {
        t.Run(tt.name, func(t *testing.T) {
            t.Parallel()

            err := parseArgs(&config{}, tt.args, io.Discard)
            if err == nil {
                t.Fatal("parseArgs() = nil, want error")
            }
        })
    }
}
```

For the `parseArgs`'s `stderr` parameter, we pass `io.Discard`. This is perhaps the most simple `Writer` implementation. It throws away the bytes sent to its `Write` method. We use it here to avoid cluttering the test log output with usage messages when parsing fails.

For the first test, we pass all the flags in a string slice and verify if we get the correct flag values. For the second test, we pass invalid flags and expect an error from the function.

We've written two tests to test the parser with valid and invalid inputs. Feel free to add other test conditions, measure the coverage, and extend this test. We can better learn when we do rather than just read. Practice makes perfect. Let's run these tests.

```
$ go test
PASS
```

Exercise

Add more test cases to both tests to increase the test coverage.

6.1.5 Wrap up

Since the `run` function operates on its input instead of globals, it has no side effects. This allows us to test it in parallel without potential race conditions. That's why we designed `run` to take an `env` as input instead of depending on the global state. This ensures that there are no dependencies between parallel invocations of our CLI logic. Let's wrap up.

- Keep the `main` function small and delegate its duty to another function.
- `io.Writer` interface abstracts writing to an output stream.
- `*bytes.Buffer` and standard streams, like `os.Stdout` are an `io.Writer`.
- Avoid calling `os.Exit` out of `main` to avoid surprises when testing.
- `SetOutput` redirects a flag set's output to an `io.Writer`.
- Use zero values when possible to simplify code.

6.2 Using build tags

Not every program's tests can run as quickly as the hit tool's tests.

Sometimes we want to skip long-running tests, especially when they involve long-running operations. Here are some common strategies to use when faced with such situations:

- Optimize our code and tests, then rerun them.
- Use the `testing.Short` function.
- Use the `t.SkipNow` method.
- Use a build tag.

Let's first use the second strategy. `Short` returns true once we run tests with the short flag:

We can check for it in the test as follows and opt to skip running the test if necessary:

```
func TestRun(t *testing.T) {
   if testing.Short() {
       t.SkipNow()
   }
   ...
}
```

Using the combination of `Short` and `SkipNow` can be cumbersome when managing many tests. Instead, build tags offer a streamlined alternative. For instance, we can prepend the following build tag to each test file that we don't want to run:

```
//go:build cli
package main
...
```

Warning

Avoid adding spaces after the double slashes. It's not "// go:build cli". Otherwise, it would be an ordinary comment. It becomes a build directive without spaces.

We should explicitly pass the cli tag as follows when we want to run the tests in those files:

```
$ go test -tags=cli
```

Conversely, to exclude tests with the specified build tag, we can use a negation:

```
//go:build !cli
```

Build tags are commonly used to distinguish unit tests from integration tests. They also help conditionally compile code for specific platforms, help manage different development environments, and toggle feature availability. To dive deeper into build tags, visit the Go documentation on Build Constraints: pkg.go.dev/cmd/go#hdr-Build_constraints.

Exercise

Add a build tag to the high-level tests file. Run them with and without the tag.

6.3 Cross-compiling

Our last topic is cross-compiling Go code.

Go shines in developing cross-platform programs. We can write code once and build it for major operating systems using the following environment variables:

- `GOOS`: Sets the target operating system.
- `GOARCH`: Sets the target machine's architecture.

They're set to the current machine's configuration by default, which we can see as follows.

```
$ go env GOOS GOARCH
```

```
darwin arm64    #A
```

Let's list all the operating systems and architectures we can compile our programs.

```
$ go tool dist list
```

```
aix/ppc64        freebsd/riscv64  linux/mips64le  openbsd/arm
android/386      illumos/amd64    linux/mipsle    openbsd/arm64
android/amd64    ios/amd64        linux/ppc64     openbsd/ppc64
android/arm      ios/arm64        linux/ppc64le   plan9/386
android/arm64    js/wasm          linux/riscv64   plan9/amd64
darwin/amd64     linux/386        linux/s390x     plan9/arm
darwin/arm64     linux/amd64      netbsd/386      solaris/amd64
dragonfly/amd64  linux/arm        netbsd/amd64    wasip1/wasm
freebsd/386      linux/arm64      netbsd/arm      windows/386
freebsd/amd64    linux/loong64    netbsd/arm64    windows/amd64
freebsd/arm      linux/mips       openbsd/386     windows/arm
freebsd/arm64    linux/mips64     openbsd/amd64   windows/arm64
```

For example, let's pick "linux/amd64" from this list. We must set the environment variables.

```
$ GOOS=linux GOARCH=amd64 go build -o hit
```

This creates an executable binary named "hit" in the current directory, which we can deploy to a Linux machine powered by an AMD 64 CPU and run.

Exercise

Compile the HIT tool to other operating systems, deploy, and run.

We've built our command-line tool from the ground up and put it through its paces with tests. The next chapter dives into building and integrating the hit package, where we'll tackle making concurrent HTTP requests to supercharge our tool's capabilities.

6.4 Summary

- Dependency injection via interfaces helps achieve testable code. Still, as with everything, there are trade-offs, and we shouldn't go too far in making every function take interfaces. We'll see when and when not to use interfaces later.
- High-level tests ensure overall program function, while low-level tests pinpoint specific issues, both critical for comprehensive test coverage.

7 Structuring Concurrent Pipelines

This chapter covers

- Designing convenient-to-use package APIs.
- Structuring and writing a concurrent HTTP client.
- Diving into the concurrent pipeline pattern.

We'll write a concurrent client package called `hit` (the HIT client). This package simulates sending concurrent requests to an HTTP server and measures the server's performance. This simulation approach will allow us to explore concurrency more deeply without getting bogged down in the details of handling HTTP requests (which we'll dive into in the next chapter).

The command-line tool we crafted previously will provide an interface for the HIT client.

Note

Find the source code of this chapter at the link: https://github.com/inancgumus/gobyexample/tree/main/hit

7.1 Laying out the groundwork

We'll first explore the `hit` package's (the HIT client) design and implement the package. Then, we'll dive into the concurrent pipeline pattern, a technique used to structure concurrent code into a series of concurrent stages or operations. It has many benefits:

- Maintainability: Easier to fix and update.
- Readability: Makes code clear and easy to follow.
- Reusability: Parts can be used again in different places (i.e., programs).

Let's design an easy-to-use and maintainable API for the HIT client.

What makes a good API?

API is everything exported from a package, which is critical because that's what other packages depend on and use. Here are some guidelines for creating a good API:

- Synchronous by default.
- Avoids unnecessary abstractions.
- Straightforward to use and avoids confusion.
- Composable. Allows users to use it in creative ways.

The API should be straightforward for beginners to use yet offer more experienced users enough knobs to twiddle. It should trust programmers, not babysit them (this is one of the Go tenets: be explicit and trust in programmers). So they can fine-tune the API behavior.

We'll follow these principles while developing the `hit` package's API.

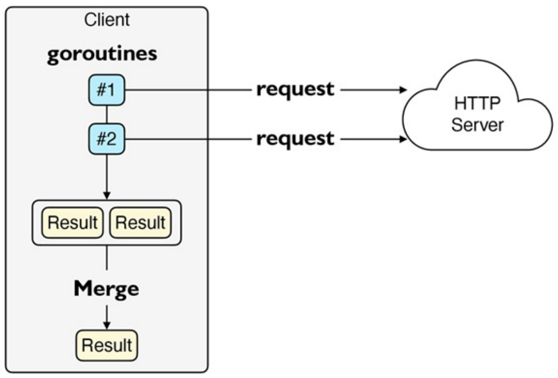
Overview

As Figure 7.1 illustrates, the `hit` package consists of two struct types:

- `Client` orchestrates sending requests in parallel and collecting performance results.
- `Result` is a performance result, such as how long the target server takes to respond, errors received, the HTTP status code, etc.

A request contains information about how and where to send HTTP requests. For instance, a request value can store the target URL we want to perform a performance test against.

Figure 7.1 Sending requests to the target server, collecting and merging results.



The HIT client sends requests in parallel using goroutines to a target URL, collects performance results, and merges the results into a single result as a summary (so we might better understand the target server's overall performance from the summarized results).

Since we looked at the overall design, we can implement the client and integrate it into the HIT tool. Initially, the client won't be concurrent and will simulate sending requests. Yet this will be a good starting point for us to develop the client throughout the chapter later.

The directory structure

Before starting, let's review the directory structure.

```
.                   -> HTTP performance measurement logic (packag

└── client.go       -> The Client type's implementation
└── client_test.go  -> Tests for the Client type
└── hit.go          -> Higher-level API for the Client type    #A
└── hit_test.go     -> Tests for the higher-level API    #A
└── pipe.go         -> Concurrent pipeline helpers        #B
└── result.go       -> The Result type
└── cmd
    └── hit         -> HIT CLI tool (package main)
```

We'll implement `Client`, `Result`, and the concurrent pipeline in this chapter. The actual sending of HTTP requests will have to wait for the next chapter to focus on concurrency.

7.1.1 Result

Let's look at the next listing for the `Result` type.

Listing 7.1 Implementing the Result type (result.go)

```
package hit


// Result is an [http.Request]'s performance result.    #A
// Its zero value is useful.                            #B
type Result struct {
    RPS      float64       // RPS is the requests per second
    Requests int           // Requests is the number of requests
    Errors   int           // Errors is the number of errors occu
    Bytes    int64         // Bytes is the number of bytes downlo
    Duration time.Duration // Duration is a single or all request
    Fastest  time.Duration // Fastest request result duration amo
    Slowest  time.Duration // Slowest request result duration amo
    Status   int           // Status is a request's HTTP status c
    Error    error         // Error is not nil if the request is
}
```

Each request will return a result whether the request succeeds or not. Once we merge all the results, we can see the total number of requests, errors, the number of bytes transferred, and so on. We'll use the `Duration` field to calculate the fastest and slowest responses. All of these will help users to see the performance of a target HTTP server.

Now that we've declared `Result`, let's move on to the `Merge` and `Finalize` methods that will help us combine the results and produce a summarized result. See the next listing.

Listing 7.2 Implementing Merge and Finalize (result.go)

```
// Merge merges this [Result] with another.
func (r Result) Merge(other Result) Result {
    r.Requests++
    r.Bytes += other.Bytes

    if r.Fastest == 0 || other.Duration < r.Fastest {
        r.Fastest = other.Duration
    }
    if other.Duration > r.Slowest {
        r.Slowest = other.Duration
    }
    if other.Error != nil || other.Status != http.StatusOK {
        r.Errors++
    }

    return r
}

// Finalize calculates the total duration and RPS.
func (r Result) Finalize(total time.Duration) Result {
    r.Duration = total
    r.RPS = float64(r.Requests) / total.Seconds()
    return r
}
```

Each method does one thing well: `Merge` combines individual request metrics, while `Finalize` computes the aggregate metrics that depend on the complete set of requests.

Let's see an example of why we need two separate methods for combining results. Say we run the following goroutines, each making a request and finishing some seconds later.

```
Goroutine #1 starts a request at 00:00 seconds and finishes at 00
Goroutine #2 starts a request at 00:01 seconds and finishes at 00
Goroutine #3 starts a request at 00:02 seconds and finishes at 00
Total duration:                  4 seconds
RPS:                             1.3 requests per second
```

If we look at each request's start and end times, the total period from the start of the first request to the end of the last is *4 seconds* because each goroutine runs in parallel. However, the total duration would look this if `Merge` were to add up the individual durations:

```
Goroutine #1 request duration: 2 seconds
Goroutine #2 request duration: 2 seconds
Goroutine #3 request duration: 2 seconds
Sum of durations:              6 seconds
RPS:                           2 requests per second
```

Adding up these gives us *6 seconds*. It isn't the time span of the whole operation since the requests were processed in parallel, not sequentially. The correct total elapsed time is from the start of Goroutine #1's request to the end of Goroutine #3's request, which is 4 seconds.

Note

We could calculate the total duration and RPS in `Merge`. But it wouldn't be this simple.

To wrap up, because we'll run goroutines in parallel, we'll use `Finalize` to properly calculate the total elapsed time and the Requests Per Second (RPS). `Finalize` calculates the metrics based on the overlapping time periods, providing an accurate measurement.

Note

You might wonder: "Aren't goroutines already parallel? What does it mean to run goroutines in parallel?". Goroutines are concurrent but not necessarily run in parallel. For instance, recall running tests in parallel from Section 4.3. Although each test runs in a goroutine, unless we call the `Parallel` method, Go test runs a single test at a time.

Dive deep: Why don't we use pointers?

You might wonder why we don't use pointers to modify the current `Result` but instead return a new one. After each request, we'll return a new `Result`. In Go (and perhaps in other languages), returning a pointer is typically a costly operation. When we unnecessarily return a pointer, the compiler might need to make a heap allocation for the returned value.

Moreover, because we'll be passing results to a concurrent pipeline (soon), we don't want the garbage collector to keep track of millions (maybe more) of results throughout our program.

All these might put an unnecessary load on the garbage collector. We're helping the garbage collector by using a value type (`Result`) instead of a pointer type (`*Result`).

This approach might produce less garbage for the collector to track and collect.

Printing results

We've declared `Result`. Our next goal is to print a result. This will be useful when printing the overall performance result or debugging. See the next listing.

Listing 7.3 Printing a result (result.go)

```
// Fprint writes [Result] to an [io.Writer].
func (r Result) Fprint(out io.Writer) {
    p := func(format string, args ...any) {        #A
        fmt.Fprintf(out, format, args...)           #A
    }                           #A

    p("\nSummary:\n")
    p("\tSuccess    : %.0f%%\n", r.successRatio())
    p("\tRPS        : %.1f\n", r.RPS)
    p("\tRequests   : %d\n", r.Requests)
    p("\tErrors     : %d\n", r.Errors)
    p("\tBytes      : %d\n", r.Bytes)
    p("\tDuration   : %s\n", round(r.Duration))

    if r.Requests > 1 {                    #B
        p("\tFastest    : %s\n", round(r.Fastest))        #B
        p("\tSlowest    : %s\n", round(r.Slowest))        #B
    }
}

// String returns result as string.
func (r Result) String() string {                #C
    var s strings.Builder                    #C
    r.Fprint(&s)                        #C
    return s.String()                    #C
}

func (r Result) successRatio() float64 {            #D
    rr, e := float64(r.Requests), float64(r.Errors)        #D
    return (rr - e) / rr * 100                #D
}

func round(t time.Duration) time.Duration {        #E
    return t.Round(time.Microsecond)            #E
}                              #E
```

Let's go over this code.

- `Fprint` prints the current result to an `io.Writer` using `Fprintf`.
- `String` satisfies the `Stringer` interface (for ease of printing a result).
- `String` uses `strings.Builder` to allow `Fprint` to efficiently generate a summary.
- `round` and `successRatio` are helpers for `Fprint`.

Inside `Fprint`, we use a helper closure, `p`, to keep `Fprint` slightly concise (this pattern is also used in Go's standard library). Like `Fprintf`, `p` takes a *format string* and *variadic* input values of any type. `args...` unpacks the variadic input and passes each element to `Fprintf`.

Note

A variadic function receives a slice with the same backing array when we pass a slice to the function using ellipsis (e.g., Fprintf(out, format, args...)). This means the function can modify the elements in the slice's backing array. Example: https://go.dev/play/p/Lz0JpSksyQV

`round` isn't a method, whereas `successRatio` is. This is because `round` doesn't use the `Result`'s fields, while the latter does. We usually declare a function if the operation is *pure* (e.g., always produces the same result with the same inputs) and doesn't require a preserving state. The opposite *might* also be true for a method.

Making the Result type a Stringer

Sometimes getting a result as a string can be more convenient.

Since most methods in Go's standard library and other packages in the wild support the `String` method, `Result` provides a `String` method that uses the result's `Fprint` method.

The `strings` package's `Builder` type is a `Writer` (implements the `Write` method). Since `Fprint` can write to a `Writer`, we can also write to a `Builder`.

The following `String` method writes the result to the internal buffer of the `strings` package's `Builder` type, and returns the buffer's content as a string:

```
func (r Result) String() string {
    var s strings.Builder
    r.Fprint(&s)
    return s.String()
}
```

The `fmt` package's printing methods can recognize if a value we want to print is a `Stringer` (implements the `String` method) and seamlessly call the method.

For example, we can now print the result as follows:

```
var sum Result
...
fmt.Fprint(os.Stdout, sum)
```

We could also print the result using other functions such as `Print` as follows:

```
fmt.Print(sum)
```

Demonstration

We'll continue from where we left off in the previous chapters. When we run the HIT tool, it'll automatically run the code we type in `runHit`. In the next listing, we're simulating receiving results from a few requests, merging the results, and printing a performance result.

Listing 7.4 Using Result (cmd/hit/hit.go)

```
package main

import (
    ...

    "github.com/inancgumus/gobyexample/hit"
)

func runHit(e *env, c *config) error {
    var sum hit.Result

    sum = sum.Merge(hit.Result{
        Bytes:    1000,
        Duration: time.Second,
        Status:   http.StatusOK,        #A
    })

    sum = sum.Merge(hit.Result{
        Bytes:    1000,
        Duration: time.Second,
        Status:   http.StatusOK,        #A
    })

    sum = sum.Merge(hit.Result{
        Duration: 2 * time.Second,
        Status:   http.StatusTeapot,     #B
    })

    sum = sum.Finalize(2 * time.Second)    #C

    sum.Fprint(e.stdout)

    return nil
}
```

The first two requests take a second to complete, while the last takes two seconds. Since we assume the requests run in parallel, we pass two seconds to `Finalize`. Let's see it in action.

```
$ go run ./cmd/hit http://example.com
 __  __     __     ______
/\ \_\ \   /\ \   /\__  _\
\ \  __ \  \ \ \  \/_/\ \/
 \ \_\ \_\  \ \_\    \ \_\
  \/_/\/_/   \/_/     \/_/

Sending 100 requests to "http://example.com" (concurrency: 1)

Summary:
        Success    : 67%
        RPS        : 1.5
        Requests   : 3
        Errors     : 1
        Bytes      : 2000
        Duration   : 2s
        Fastest    : 1s
        Slowest    : 2s
```

RPS is 1.5 because the requests finished in two seconds. The fastest request was measured as one second, while the slowest was two seconds (the one with `StatusTeaPot`). The last result caused the error counter to increment since it was `StatusTeapot`. Recall that `Result` considers a request only successful if the HTTP status code is OK (200).

HTTP Status Code 418: Teapot

HTTP status code 418 started as an April Fool's joke, referencing Hyper Text Coffee Pot Control Protocol (datatracker.ietf.org/doc/html/rfc2324). People wanted to remove status code 418, and others started the "Save 418 Movement" (save418.com).

7.1.2 Client

`hit.Client` will initially be synchronous and simulate making requests (instead of sending HTTP requests to an HTTP server (this feature will have to wait for the next chapter)).

This will allow us to make a quick start. Once we implement the concurrent pipeline pattern in the following section, `hit.Client` will spawn goroutines and orchestrate concurrent requests. Goroutines will use channels to communicate with each other.

Recall from the earlier chapters (Section 2.4) that channels are concurrency-safe strongly-typed queues that allow goroutines to communicate and synchronize.

Implementing the HIT client

As shown in the following listing, `Client` has a straightforward API that combines results and returns a single result. It gets each result from `Send` (which we'll implement soon).

Listing 7.5 Implementing the Client type (client.go)

```
package hit

import (
    "net/http"
    "time"
)

// Client sends HTTP requests and returns an aggregated result.
type Client struct {
    C   int // C is the concurrency level
    RPS int // RPS throttles the requests per second
}

// Do sends n HTTP requests and returns an aggregated result.
func (c *Client) Do(r *http.Request, n int) Result {
    t := time.Now()            #A

    var sum Result
    for range n {
        // the final report will include errors encountered.
        // skipping the error result here.
        result, _ := Send(r)        #B
        sum = sum.Merge(result)        #C
    }

    return sum.Finalize(time.Since(t))    #D
}
```

`Do` simulates sending HTTP requests, measures the total duration of all requests, and returns a summary. It takes `http.Request` as an input parameter. We'll soon use `Request` as a template and clone it for each goroutine. Next, let's implement the `Send` function.

Note

`http.Request` carries information such as which URL to send a request, HTTP headers, and cookies to use to send a request. See https://pkg.go.dev/net/http#Request

Simulating HTTP requests

The following listing shows the `Send` function's implementation.

Listing 7.6 Implementing the Send function (hit.go)

```
package hit

import (
    "io"
    "net/http"
    "time"
)

// Send an HTTP request and return a performance result.
func Send(r *http.Request) (Result, error) {
    t := time.Now()            #A

    time.Sleep(100 * time.Millisecond)    #B

    result := Result{
        Duration: time.Since(t),        #C
        Bytes:    10,
        Status:   http.StatusOK,        #D
    }

    return result, nil
}
```

The `Send` function doesn't do anything useful yet and simulates making a fake request and returns a fake result. It sleeps for one hundred milliseconds to simulate making an HTTP request. The returned result is always successful because its status is `OK`, which equals `200`.

Then again, it'll let us see the HIT client in action.

Tip

We export `Send` to let other packages build their clients if they wish (as `Client` does). For instance, they can run `Send` and use `Result`'s methods to calculate performance results. That's also why we return an error from `Send` while the error can already be saved in the `Error` field. This makes it explicit that `Send` returns an error instead of burying the error in `Result`. So the callers can check the error to decide whether `Send` has failed.

Integrating the HIT client into the HIT tool

It's time to integrate the HIT client into the HIT tool and see it in action.

Doing so will help us keep improving the client throughout the chapter.

The next listing updates the `runHit` function we added in the last chapter. `runHit` passes a new request to the client to simulate sending requests to a server and prints the result.

Listing 7.7 Integrating the HIT client (cmd/hit/hit.go)

```
func runHit(e *env, c *config) error {
    handleErr := func(err error) error {
        if err != nil {
            fmt.Fprintf(e.stderr, "\nerror occurred: %v\n", err)
            return err
        }
        return nil
    }

    request, err := http.NewRequest(http.MethodGet, c.url, http.N
    if err != nil {
        return handleErr(fmt.Errorf("new request: %w", err))
    }

    client := &hit.Client{
        C: c.c,                               #B
    }
    sum := client.Do(request, c.n)                      #C
    sum.Fprint(e.stdout)                           #C

    return nil
}
```

The `handleErr` is a helper to print and return errors to the HIT tool to handle. We consider the `runHit` function like a mini program that handles its own errors and prints them to the standard error. Returning errors from this function is only for testing purposes.

Then, we create a new `GET` request that doesn't have a request body (`http.NoBody`) with the URL provided to us from the HIT tool (`c.url`). `NewRequest` returns an error if one of the provided arguments is invalid. Lastly, we pass the request to `Do` to send requests (with `c.n`).

Note

`NewRequest` takes an `io.Reader` for the request's body. Setting this parameter is useful when we want to send the body (e.g., a POST request). `NoBody` is a `Reader` that does nothing.

We've integrated the first version of the HIT client into the HIT tool. Let's take it for a ride.

The following command simulates sending ten requests to the URL. Since the HIT client doesn't yet use parallelism, the sending of these requests will be sequential.

```
$ go run ./cmd/hit -n 10 -c 10 http://example.com            #A
 __  __     __     ______
/\ \_\ \   /\ \   /\__  _\
\ \  __ \  \ \ \  \/_/\ \/
 \ \_\ \_\  \ \_\    \ \_\
  \/_/\/_/   \/_/     \/_/

Sending 10 requests to "http://example.com" (concurrency: 10)

Summary:
        Success    : 100%
        RPS        : 9.9
        Requests   : 10
        Errors     : 0
        Bytes      : 100
        Duration   : 1.012753s
        Fastest    : 100.18ms
        Slowest    : 103.436ms
```

Underneath, `Send` simulates that each request takes 100 milliseconds, RPS (requests per second) becomes about ten for ten requests (simulating that the server can serve 10 RPS).

7.1.3 Wrap up

This conclude the first version of the HIT client. We'll make it concurrent in the next section.

- `Client` sends requests and returns an aggregated `Result`.
- `Result` is an HTTP request's performance result.
- `Merge` merges a `Result` with another.

- `Finalize` sets a `Result`'s total duration.
- `Fprint` prints a `Result` to the console.
- `http.Request` carries request details, such as URL, HTTP headers, etc.
- http.NewRequest returns a new http.Request.

7.2 Concurrent pipeline pattern

Sending sequential requests to a server wouldn't reflect its real performance. One request could take seconds, while the server could simultaneously serve others. Previously, we ran the HIT tool and made ten requests in a second (RPS=10), but each was made sequentially.

Using ten goroutines could increase the requests per second (RPS) to one hundred.

By leveraging concurrency, we can gather results faster. Instead of sending one request and waiting for its response before sending the next, we can dispatch multiple requests simultaneously and wait for the responses concurrently. This will increase the throughput.

Since each request's processing time likely exceeds the minimal communication delay between goroutines—*typically less than a microsecond*—we can benefit from parallel execution without significant inter-goroutine communication overhead (it's not free).

It's time to design and implement the concurrent pipeline and integrate it with `Client`.

Note

Avoid overusing goroutines if the time expected for goroutine channel communication is longer than the tasks themselves. Excessive overhead from communicating between too many goroutines could negate the benefits of parallel execution.

7.2.1 What is a concurrent pipeline?

Concurrency is structuring a program as independently executing components. The concurrent pipeline perfectly fits the bill. Think of it as a Unix pipeline or an assembly line.

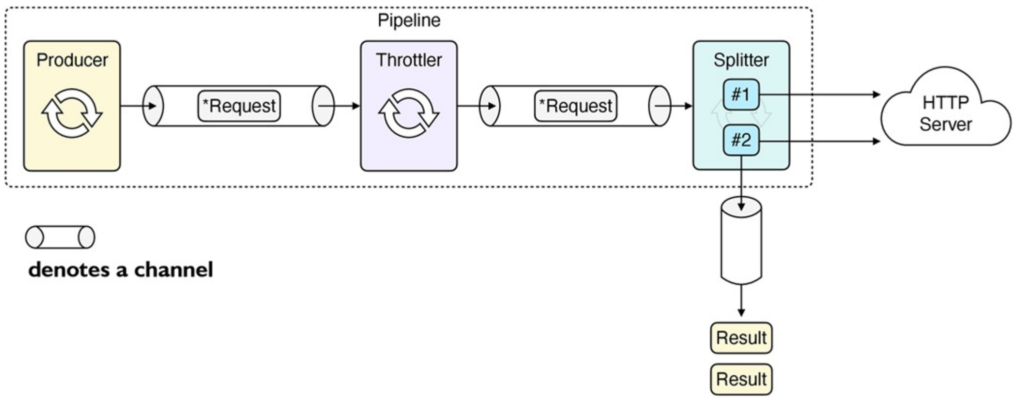
A concurrent pipeline is a modular pattern consisting of stages. Each stage operates concurrently, transmitting messages to the subsequent stage through channels.

As Figure 7.2 shows, the HIT client's concurrent pipeline has three concurrent stages.

- **Producer** clones and distributes requests to the next stage.
- **Throttler** slows down the sending of requests if desired.
- **Splitter** spawns goroutines to send HTTP requests to a server in parallel.

Goroutines send results to the splitter's output channel. While requests are in progress, we can listen to the splitter's channel and receive and merge each result as they come.

Figure 7.2 The concurrent pipeline's overall design.



So, everything starts with the producer. *The throttler is optional.*

The splitter receives requests either from the producer or throttler and sends parallel requests. The producer stage is the pipeline's entry point and actuator. The rest listen for requests from the producer's output channel (either throttler or splitter).

They stop working when the producer closes the channel.

Note

The producer generates requests, but only the splitter sends them to a server.

Each stage in a concurrent pipeline is typically run by a separate goroutine (allowing for concurrent operation), and they communicate via channels.

Without channels, stages would be deadlocked unless we use buffered channels. However, even if we use buffered channels, the stages wouldn't work in a lockstep manner (as goroutines can continue to execute without the immediate coordination we expect) and might be hard to debug. Synchronous channels are easier to debug and comprehend.

So, we'll keep it simple and effective.

Note

One of the strong aspects of using a concurrent pipeline is that we can easily add and remove stages and compose different pipelines without changing stage code.

http.Request

Before getting started, let's look at what the `http.Request` type looks like. Recall that the producer will produce and pass requests to other stages for consumption.

```
// A Request represents an HTTP request.
type Request struct {
    // Method specifies the HTTP method (GET, POST, PUT, etc.).
    Method string

    // URL specifies the URL to access.
    URL *url.URL

    // Body is the request's body.
    Body io.ReadCloser
    // ..other fields...
}
```

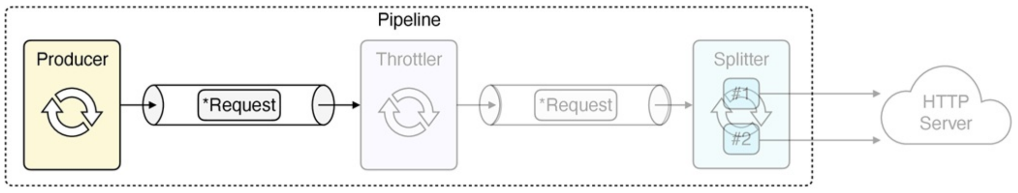
The `Method` field determines the HTTP method, like GET. We'll use the `URL` field to specify the target server URL we want to measure its performance. We won't actually use the `Body` field but demonstrated here because we must provide a body while creating a new request.

A request cannot make any requests to a server. It only stores request details. For that, in the next chapter, we'll need to use the `http.Client` type, which we can pass an `*http.Request` to send a request to a URL specified in the request's `URL` field.

7.2.2 Producer stage: Generating requests

Let's start implementing the producer stage. Once we implement all the stages, we'll connect them to bring the concurrent pipeline to life. See Figure 7.3 to see where we are.

Figure 7.3 The producer stage produces and sends requests to the next stage.



The next listing shows the producer stage.

Listing 7.8 Implementing the producer stage (pipe.go)

```
package hit

import "net/http"

// Produce calls fn n times and sends results to out.
func Produce(out chan<- *http.Request, n int, fn func() *http.Req
    for range n {
        out <- fn()
    }
}
```

```
// produce runs Produce in a goroutine.
func produce(n int, fn func() *http.Request) <-chan *http.Request
    out := make(chan *http.Request)
    go func() {
        defer close(out)            #A
        Produce(out, n, fn)
    }()
    return out                #B
}
```

We have two functions: `Produce` and `produce`.

- `Produce` produces (using `fn`) and sends `n` requests to its output channel.
- `produce` makes an output channel (`out`), runs `Produce` in a goroutine, and returns the channel. Once `Produce` finishes and returns, `produce` closes the channel.

Although we return the channel, the `Produce` goroutine will keep running. This allows us to call `produce` once, get the channel, and pass it to the rest of the stages, like the throttler.

Note

For those familiar with asynchronous functions in JavaScript, this is similar to promises. `create_task` in Python, `CompletableFuture.supplyAsync` in Java, and `Task.Run` in C#.

The next stage will listen to the channel for new requests until the channel is closed. The goroutine will close the channel after delivering all the requests to tell the listeners to stop listening. Otherwise, the listeners would be blocked forever, waiting for more requests.

Tip

Closing a channel is a way to signal other goroutines that sending is finished. It's not a resource-reclaiming mechanism. We should close a channel solely when we want to inform the receiving goroutines. Since sending a value to a closed channel panics, it's idiomatic that the channel owner closes it to ensure that all values are sent.

Let's go over the producer stage's code in more detail.

Let's start with `Produce`:

```
func Produce(out chan<- *http.Request, n int, fn func() *http.Req
```

It takes a *send-only* channel to send the generated requests:

```
out chan<- *http.Request
```

Directional channels (receive-only and send-only) make introducing bugs less likely and show the intentions of what to do (and what we cannot and should not do) with a channel.

The last input is a function to leave the decision of how to produce requests to the caller:

```
fn func() *http.Request
```

This allows modularity in our design. For example, the caller can pass a function that logs the generated request for later analysis before returning it from this function. However, we'll only clone and return a request. This is the bare minimum for the pipeline to work. For instance:

```
Produce(output, 100, func() *http.Request {

    return request.Clone(...)    #A
})
```

Then, the rest of the stages can work on this cloned request to do their work.

The orchestrator pattern

The `produce` function in Listing 7.8 is the orchestrator that runs the producer's main logic (`Produce`) in a separate goroutine. Separating a function that contains the main logic (`Produce`) from the goroutine that would run it (`produce`) is an effective pattern.

For example, the caller can use a buffered channel to buffer all the request values into the channel (hence into memory) upfront and immediately close the channel and free the producer. The producer's output channel would be filled with requests even if the producer is gone. And the rest of the stages would keep working until they are out of request values.

This can be helpful in, say, reading files to process them, giving the caller a chance to close the file and the producer while the data that was read continues to be processed.

One last benefit is that the `hit` package's importers can create a custom client type. That's why we exported `Produce` so others can use it to create a pipeline. But `produce` is unexported because only `Client.do` would use it.

Like LEGO bricks

There's a subtle detail here that we should discuss. By accepting a channel input, `Produce` allows its callers to build their own pipeline (think of it like LEGO bricks). For instance, they might craft a custom client for specific purposes instead of using the HIT client.

Here's an example:

```
// produce 100 requests
output := make(chan *http.Request)

go hit.Produce(output, 100, func() *http.Request {
    req, _ := http.NewRequest(            #A
        http.MethodGet, "http://foo.com", http.NoBody,
    )
    req.Header.Set("X-Trace-Id", strconv.Itoa(rand.N(100)))
    return r
})

// send each request to http://foo.com
for range 100 {
    r := <-out
    go makeRequest(r)                #B
}
```

`Produce` generates one hundred HTTP requests with a random trace ID header, sending each request to the `output` channel. Then, the loop below takes these requests from the channel and sends the requests to a server (e.g., http://foo.com). Of course, this code could just loop one hundred times to send requests concurrently without using `Produce`. However, the benefit of `Produce` will come out when we combine it with other pipeline stages.

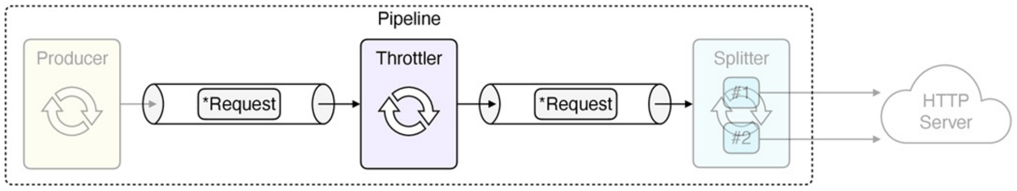
Let's continue implementing our own pipeline before going into the rabbit hole.

7.2.3 Throttler stage: Slowing down the pipeline

It's valuable to understand what happens when the target server is overloaded. But we might wish to understand the performance until it greatly drops because the server is too overloaded. So, we might want to slow down sending requests until we find an optimal rate.

We can do this with a *throttler* stage. See Figure 7.4 to see where we are.

Figure 7.4 The throttler stage slows down the delivery of requests.



See the next listing for the throttler stage's implementation.

Listing 7.9 Implementing the throttler stage (pipe.go)

```
// Throttle slows down receiving from in by delay and
// sends what it receives from in to out.
func Throttle(in <-chan *http.Request, out chan<- *http.Request,
    t := time.NewTicker(delay)
    defer t.Stop()

    for r := range in {
        <-t.C
        out <- r
    }
}

// throttle runs Throttle in a goroutine.
func throttle(in <-chan *http.Request, delay time.Duration) <-cha
```

```
    out := make(chan *http.Request)
    go func() {
        defer close(out)
        Throttle(in, out, delay)
    }()
    return out
}
```

Again, we use the orchestrator pattern.

- `Throttle` is the throttler stage's main logic.
- `throttle` runs `Throttle` in a goroutine.

Unlike `Produce`, this time, `Throttle` takes input and output channels:

Throttle(in <-chan *http.Request, out chan<- *http.Request, delay time.Duration)

This is because, unlike the producer, the throttler is an intermediate stage. It's a stage that receives input from a previous stage (i.e., the producer stage) and passes its output to a subsequent stage (i.e., the splitter stage). It's a middleperson in the flow of requests.

Returning to the throttler stage, once we connect it to the pipeline, it'll receive requests from the producer stage. The throttler stage's input channel will be the producer stage's output channel and the throttler stage's output channel will be the splitter stage's input channel.

```
producer() output channel
           ==============
                |||
                vvv
  throttler(input channel) output channel
                           ==============
                                |||
                                vvv
                   splitter(input channel)
```

Ticker

The last input is a time duration that specifies how long to wait before sending requests to the output channel for each request received from the input channel. Depending on this input parameter, the throttler stage creates a `Ticker` to receive periodic ticks to slow down delivering the request values to the next stage. `Ticker` looks like this:

```
// A Ticker holds a channel that delivers "ticks" of a clock
// at intervals.
type Ticker struct {
    C <-chan Time // The channel on which the ticks are delivered
    // ...
}
```

We use the `NewTicker` function to create a new `Ticker`:

```
// NewTicker returns a new Ticker containing a channel that will
// the current time on the channel after each tick. The period of
// ticks is specified by the duration argument. The ticker will a
// the time interval or drop ticks to make up for slow receivers.
// The duration d must be greater than zero; if not, NewTicker wi
// panic. Stop the ticker to release associated resources.
func NewTicker(d Duration) *Ticker
```

`Ticker`, once created with `NewTicker`, periodically sends a message to its channel, `C`.

The throttler waits on this channel before sending the incoming request to the next stage, slowing down the flow of requests. Although unnecessary, the throttler stops the ticker as soon as its input channel is closed (i.e., the producer's output channel). Closing the ticker's channel is no longer necessary because the compiler can automatically do that. See the link to the issue for more details: https://github.com/golang/go/issues/61542.

Imagine we set the throttler to make a request each second, and the producer gets two seconds late sending messages to the throttler. If we were to use `time.Sleep`, this operation would take three seconds (consider why that is so before reading the next notice).

Note

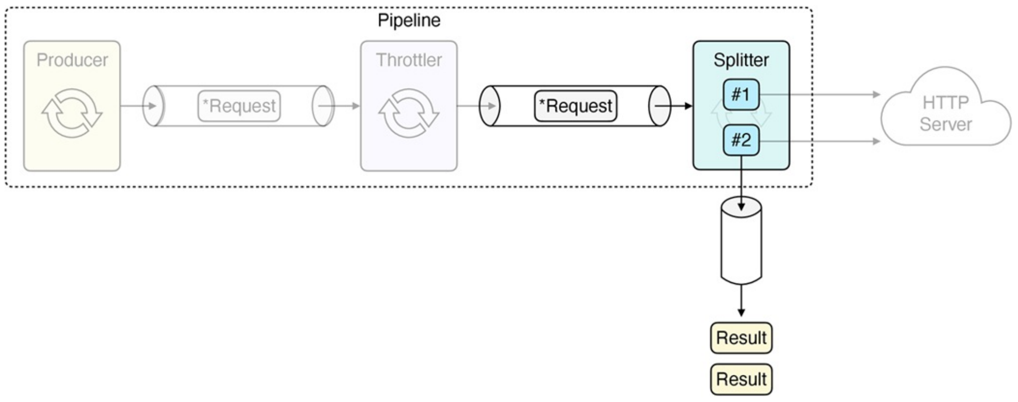
`Sleep` pauses the "current goroutine" for at least the specified `time.Duration`.

However, with `Ticker`, it would immediately tick since the producer would be already late sending messages for two seconds. `Ticker` has an internal timer that keeps ticking. So, `Ticker` attempts to tick at the configured rate, but in cases where we retrieve the ticks too slowly, it can also *drop ticks*. See the link for more details: https://pkg.go.dev/time#Ticker

7.2.4 Splitter stage: Sending requests in parallel

Sending requests to a server one at a time can be sluggish. We can make it faster by sending requests in parallel. The splitter stage splits the incoming request values among goroutines to send parallel requests. See Figure 7.5 for the last stage, the splitter.

Figure 7.5 The splitter stage sends requests in parallel to a target server.



Let's look at the splitter stage's implementation in the next listing.

Listing 7.10 Implementing the splitter (pipe.go)

```
// SendFunc is the type of the function that sends an HTTP
// request and returns a performance result.
type SendFunc func(*http.Request) Result

// Split splits the pipeline into c goroutines, each running fn w
// what split receives from in, and sends results to out.
func Split(in <-chan *http.Request, out chan<- Result, c int, fn
    send := func() {
        for r := range in {
            out <- fn(r)
        }
    }

    var wg sync.WaitGroup
    wg.Add(c)

    for range c {
        go func() {
            defer wg.Done()
            send()
        }()
    }

    wg.Wait()
}

// split runs Split in a goroutine.
func split(in <-chan *http.Request, c int, fn SendFunc) <-chan Re
    out := make(chan Result)
    go func() {
        defer close(out)
        Split(in, out, c, fn)
    }()
    return out
}
```

Here, we see the orchestrator pattern again.

- `Split` is the splitter stage's main logic.
- `split` runs `Split` in a goroutine.

`Split` takes a receive-only input channel to receive requests, a send-only channel to send request results, a concurrency level, and a function that sends a request to a server:

```
Split(in <-chan *http.Request, out chan<- Result, c int, fn SendF
```

Unlike other stages, `Split` takes a `Request` channel but returns a `Result` channel. This is because the splitter stage is the pipeline's terminal stage.

But, we don't see the logic of sending HTTP requests inside the function. How does it work?

`Split` is a reusable function that doesn't know how to send HTTP requests.

Its role is to split the pipeline into multiple lanes depending on the `c` input parameter (the concurrency level). Each lane is run by a goroutine in parallel. Inside the function, we see a helper, `send`:

```
send := func() {
    for r := range in {    #A
        out <- fn(r)    #B
    }
}
```

It keeps receiving incoming requests from the splitter's input channel (`in`) and sends the performance results to the output channel (`out`) using the `fn` function.

The `send` function will run by separate goroutines:

```
for range c {
    go func() {
        defer wg.Done()
        send()
    }()
}
```

Each goroutine will call `send` *about* `N/C` (the number of requests/concurrency level) times to share the total amount of work. Of course, this share ratio depends on how long each unit of work takes (processing an HTTP request), and we don't know the exact ratio beforehand.

We'll pass the `Send` function we developed earlier to `Split` for the goroutines to send HTTP requests. Let's recall the `Send` function:

```
// Send an HTTP request and return a performance result.
func Send(r *http.Request) (Result, error) {
    ...

    result := Result{
        Duration: time.Since(t),
        Bytes:    10,
        Status:   http.StatusOK,
    }

    return result, nil
}
```

For example, by passing `Send` to `Split`, goroutines can send HTTP requests (soon):

```
Split(..., func(r *http.Request) Result {
    result, _ := Send(r)
    return result
})
```

We're skipping the error handling with a blank-identifier because the pipeline will errors (each `Result` saves the error into their `Error` field for later analysis).

WaitGroup

Inside the `Split` function, we see a type we haven't seen before: `WaitGroup`.

```
var wg sync.WaitGroup    #A


wg.Add(c)        #B
for range c {        #C
    go func() {        #C
        defer wg.Done()    #D
        send()
    }()
}
wg.Wait()        #E
```

`WaitGroup` allows a goroutine to wait for all the other goroutines to finish their work.

It's similar to a counting semaphore:

- `Add` increments the semaphore's counter.
- `Done` decrements the counter (calls `Add(-1)`).
- `Wait` blocks the goroutine until the counter reaches zero.

We use `WaitGroup` to wait for all goroutines to send requests. Thanks to `WaitGroup`, `Split` won't return before all goroutines finish their work. Otherwise, the `go` statement would launch the goroutines and `Split` would return without waiting for the goroutines.

This would cause `split` to close the output channel and cause a data race issue.

Note

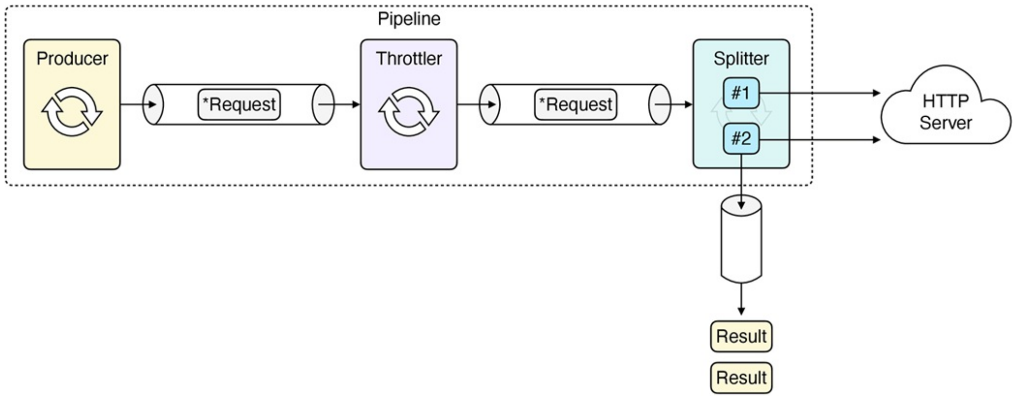
See hit/pipe.go in the book's repository for an alternative implementation (SplitLimit).

7.2.5 Connecting the stages

Our goal has been to send requests efficiently by processing them in parallel using a concurrent pipeline where each stage is an independently running concurrent component. Since we've finished implementing all the stages, it's time to connect them.

Let's recall the pipeline before getting into the implementation. See Figure 7.6.

Figure 7.6 The concurrent pipeline where every stage is enabled.



Bringing the pipeline to life

In the next listing, we're connecting the producer, throttler, and splitter stages to process parallel requests. Then, we're receiving performance results from the splitter stage.

Listing 7.11 Connecting the stages (client.go)

```
// Do sends HTTP requests and returns an aggregated result.
func (c *Client) Do(r *http.Request, n int) Result {
    t := time.Now()

    var sum Result                #A
    for result := range c.do(r, n) {        #A
        sum = sum.Merge(result)           #A
    }                         #A

    return sum.Finalize(time.Since(t))
}

func (c *Client) do(r *http.Request, n int) <-chan Result {
    pipe := produce(n, func() *http.Request {           #B
        return r.Clone(context.TODO())                #B
    })                                 #B

    if c.RPS > 0 {
        t := time.Second/time.Duration(c.RPS*c.concurrency())
        pipe = throttle(pipe, t)                   #C
    }

    return split(pipe, c.concurrency(), Send)           #D
}

func (c *Client) concurrency() int {
   if c.C > 0 {
       return c.C
   }
   return runtime.NumCPU()
}
```

This code is similar to the previous one we implemented in Listing 7.5 of Section 7.1.2. The difference is that we now use a concurrent pipeline, listen to the results from a channel, and merge the results to return a final performance result. The `do` method is responsible for managing the pipeline and delivering results to the caller (i.e., the `Do` method).

The `n` parameter determines the number of requests to send to the pipeline. The `C` field determines the number of goroutines to use while sending requests. Lastly, the `RPS` field throttles requests per second if it's set to a positive number (otherwise, it's defunct).

Tip

It's convenient to see what's happening in a method in one go: What `do` does is clear. Rather than burying the cloning of requests logic inside `produce`, `produce` takes a function and leaves the decision to the caller (in this case, `do`). We can also say the same for `split`. Plus, doing so allows us to change the `do` method's logic and the functions separately in the future. Lastly, the functions become more reusable (`produce` and `split`).

Let's go over the code in more detail.

The producer stage

```
pipe := produce(n, func() *http.Request {

    return r.Clone(context.TODO())
})
```

This callback deep clones the requests (all its fields), so the producer can produce identical requests to the same target URL. If request values were not cloned, there would be data race issues and clashes with other ongoing request values since each request value is stateful.

While cloning the request, we're passing `context.TODO` to the `Clone` method because it requires a `context.Context` (an interface in the `context` package):

```
// Clone returns a deep copy of r with its context changed to ctx

// The provided ctx must be non-nil.
//
// For an outgoing client request, the context controls the entir
// lifetime of a request and its response: obtaining a connection
// sending the request, and reading the response headers and body
func (r *Request) Clone(ctx context.Context) *Request
```

Here's also the `TODO` context's signature:

```
// TODO returns a non-nil, empty [Context]. Code should use conte
// it's unclear which Context to use or it is not yet available (
// surrounding function has not yet been extended to accept a Con
// parameter).
func TODO() context.Context
```

We'll detail the `context` package in the next chapter, but here's a rundown. We can use `context.TODO` when we don't know what `Context` to pass to another function that requires a `Context` (avoid passing a `nil` `Context` to a function as it might panic). We use `TODO` because we haven't yet updated the client to work with a `Context`. We'll do so in the next chapter.

Returning to the producer stage's current role, it doesn't do much work other than cloning the request. Does it have to be a concurrent stage? It doesn't have to be concurrent if that's all it does. However, we design these stages to be modular and interchangeable within the pipeline. So, it still makes sense to keep the producer as a concurrent stage.

Imagine a situation where the producer is improved to acquire real-time data on user behavior patterns from an external analytics system. This data could then be used to modify the parameters of the cloned requests. For example, the query string could be changed, or headers could be adjusted to reflect different user profiles or search behaviors. This enhancement would enable the producer to replicate various user interactions, resulting in a more comprehensive evaluation of the server's performance across diverse use cases. Each stage can also run their own internal goroutines if needed to improve their throughput.

The throttling stage

Next, let's look at the throttling stage.

```
if c.RPS > 0 {
    t := time.Second/time.Duration(c.RPS*c.concurrency())
    pipe = throttle(pipe, t)                        #A
}
```

We want to enable throttling only when the user wants to throttle requests; hence we can do so if the `RPS` field is provided. Once throttling is enabled, we pass the producer's output channel as the input channel to the throttler and get the throttler's output channel:

producer -> requests -> throttler -> wait a while -> requests -> next stage

By doing so, the channel becomes the throttler's input channel and the throttler receives request values from the producer. Then, the loop in `Do` listens to the throttler's channel instead.

The `do` function divides one second by `RPS` to determine how many requests to send per second. For example, the throttler slows down each request value by half a second if `RPS` is two (`second/2=half a second`) so that the pipeline sends two requests per second.

Then, we multiply `RPS` with `C` (`Client.concurrency()`) to adjust for concurrency.

Otherwise, the throttler would merely slow down all parallel requests to a level specified by `RPS`, and sending parallel requests wouldn't make sense. Lastly, we connect the splitter to the pipeline using the producer *or* the throttler's output channel.

The splitter stage

The splitter distributes the incoming request values to multiple goroutines.

split(pipe, c.concurrency(), Send)

We're telling the splitter stage to spawn goroutines and send requests to the target URL.

- The first argument is the input channel (requests from earlier stages).
- `c.concurrency()` determines the number of goroutines to spawn.
- The last argument, `Send`, is the `Send` function that sends HTTP requests.

Each goroutine runs `Send` to send HTTP requests and delivers the performance results.

We have connected all the stages and created a concurrent pipeline to process requests.

Setting the throttling flag

Now that we have the `RPS` field, let's set it in the next listing. Otherwise, the throttler would never be active. Setting `RPS` will let the client enable throttling in the concurrent pipeline.

Listing 7.12 Integrating the throttling (cmd/hit/hit.go)

```
func runHit(e *env, c *config) error {
    handleErr := func(err error) error {
        if err != nil {
            fmt.Fprintf(e.stderr, "\nerror occurred: %v\n", err)
            return err
        }
        return nil
    }

    request, err := http.NewRequest(http.MethodGet, c.url, http.N
    if err != nil {
        return handleErr(fmt.Errorf("new request: %w", err))
    }

    client := &hit.Client{
        C:   c.c,
        RPS: c.rps, // <---
    }
    sum := client.Do(request, c.n)
    sum.Fprint(e.stdout)

    return nil
}
```

Users can now throttle the requests.

Note

We're witnessing how straightforward and effortless it is to add a new feature once we have spent the time to design and structure the code well to begin with.

Taking the pipeline for a ride

We've connected all the stages to make a concurrent pipeline and set the throttling field. Everything is ready for sending (simulating) requests in parallel. Let's give it a try. We'll send (simulate) one thousand concurrent requests by distributing them among ten goroutines:

```
$ go run ./cmd/hit -n 1000 -c 10 http://example.com

 __  __     __     ______
/\ \_\ \   /\ \   /\__  _\
\ \  __ \  \ \ \  \/_/\ \/
 \ \_\ \_\  \ \_\    \ \_\
  \/_/\/_/   \/_/     \/_/

Sending 1000 requests to "http://example.com" (concurrency: 10)

Summary:
        Success    : 100%
        RPS        : 99.0
        Requests   : 1000
        Errors     : 0
        Bytes      : 10000
        Duration   : 10.110839s
        Fastest    : 100.008ms
        Slowest    : 104.565ms
```

Recall that `Send` simulates making a request, and each takes one hundred milliseconds. That's why the total duration is ten seconds.

Let's try the same command and send sequential requests (i.e., with one goroutine):

```
$ go run ./cmd/hit -n 1000 -c 1 http://example.com

 __  __     __     ______
/\ \_\ \   /\ \   /\__  _\
\ \  __ \  \ \ \  \/_/\ \/
 \ \_\ \_\  \ \_\    \ \_\
  \/_/\/_/   \/_/     \/_/

Sending 1000 requests to "http://example.com" (concurrency: 1)

Summary:
        Success    : 100%
        RPS        : 9.9
        Requests   : 1000
        Errors     : 0
        Bytes      : 10000
        Duration   : 1m41.151939s
        Fastest    : 100.049ms
        Slowest    : 118.681ms
```

It takes about two minutes to send one thousand requests using a single goroutine. The concurrent version is 10X faster! Note that the throttling wasn't active in the previous runs.

Let's now activate the throttler to one request per second and send concurrent requests.

```
$ go run ./cmd/hit -n 1000 -c 10 -rps 1 http://example.com

 __  __     __     ______
/\ \_\ \   /\ \   /\__  _\
\ \  __ \  \ \ \  \/_/\ \/
 \ \_\ \_\  \ \_\    \ \_\
  \/_/\/_/   \/_/     \/_/

Sending 1000 requests to "http://example.com" (concurrency: 10)

Summary:
        Success    : 100%
        RPS        : 10
        Requests   : 1000
        Errors     : 0
        Bytes      : 10000
        Duration   : 1m40s
        Fastest    : 102.021ms
        Slowest    : 121.121ms
```

This tells our pipeline to reduce the number of requests per second to one. The client adjusted the throttling to ten since there were ten goroutines (-c 10). As explained in *Section 7.2.5's connecting the stages* subsection, otherwise, the throttler would merely slow down the requests to a level specified by `RPS` (*we would make one request per second*), and sending parallel requests wouldn't make sense.

If we were not using the concurrency flag and set it to one RPS would be one too, and the total duration would almost feel like forever (well, it'd be about two minutes!).

7.2.6 Concurrency is an implementation detail

Let's recall the `Do` method.

```
type Client struct { ... }
```

```
func (c *Client) Do(r *http.Request, n int) Result
```

Notice that the `Do` method's API is synchronous even though it internally runs goroutines and uses channels. It avoids taking or returning a channel parameter. Since `Do` is synchronous, it never returns until it finishes its work. What we get from `Do` is only a `Result` value. `Do` leaves the decision of concurrency to the callers. This saves the callers from managing concurrency.

Counter example: An API that exposes concurrency

Instead, let's say `Do` were to expose concurrency details in its API:

```
func (c *Client) Do(r *http.Request, n int) <-chan Result
```

In this case, callers would need to manage concurrency themselves. It wouldn't be a good user experience for our elegant package. For instance:

```
t := time.Now()

client := &hit.Client{
    C: 10,
}

var sum hit.Result
for result := range client.Do(request, 100) {
    sum = sum.Merge(result)
}

sum = sum.Finalize(time.Since(t))
sum.Fprint(os.Stdout)
```

We're duplicating the work we have done in the `Do` method. Otherwise, it wouldn't be possible to get a performance summary. Moreover, we might never know when the caller pulls the results. They could unintentionally clog our pipeline if they prematurely stop receiving from the results channel while we have results to push in the pipeline.

Note

The pipeline processes the next set of requests once the delivered results are pulled.

Better example: An API that hides concurrency

Hide unnecessary complexity behind a synchronous API and let other people decide when to use the API concurrently. Concurrency is an implementation detail.

For instance, callers can easily turn our original synchronous API into a concurrent one by running it in goroutines themselves:

```
results := make(chan hit.Result)
for range 10 {
    go func() {
        client := &hit.Client{
            C: 10,
        }
        results <- c.Do(getNextRequest(), 100)    #A
    }()
}
for range 10 {
    fmt.Println(<-results)
}
```

Assume that the `getNextRequest` function returns an `*http.Request` with a different URL each time it's called. So, we can concurrently send requests to separate URLs.

In total, we're running ten thousand requests shared by one hundred goroutines. Then, once we retrieve a `Result` from the `results` channel, we print the performance results once.

Tip

Concurrency is an implementation detail. Let the users decide when to use concurrency or not. Design a synchronous API. The standard library's API is mostly synchronous. Internally, it might launch goroutines but hides this complexity from the package APIs (hence users).

7.2.7 Wrap up

We started the section with a client that could only send sequential requests and made it concurrent. We now have a program to send and process concurrent requests. We added the concurrency to the `hit` package without changing its API. Since *concurrency is an implementation detail*, users wouldn't notice that we made the `hit` package concurrent.

Let's summarize:

- Once we correctly structure the code, adding new features is a breeze.
- Concurrency is structuring a program as independently executing components.
- A pipeline connects a set of concurrently running stages.
- `Ticker` sends periodic intervals to a channel.
- time.NewTicker returns a new Ticker.
- `sync.WaitGroup` is like a counting semaphore that allows us to wait for goroutines.
- `WaitGroup.Add` increments the semaphore.
- `WaitGroup.Done` decrements the semaphore.
- `WaitGroup.Wait` blocks the current goroutine until the counter reaches zero.
- `Context` allows us to propagate cancellation signals across code and goroutines.
- `Context.TODO` is an uncancellable context we can use when we don't know what `Context` to use. It's typically usually used in functions that don't support `Context`.

In the next chapter, we'll start sending HTTP requests and test the client.

7.3 Exercises

1. Add a `SendFunc` field (of type `SendFunc`) to `Client` to customize request

sending.

2. Test the `Result` and `Client` types to ensure they work correctly.
3. Increase the test coverage by at least 70%.
4. Add a new timeout flag to the hit CLI for the `hit.Client.Timeout` field.
5. Add an `Errors []error` field to `Result` that accrues errors encountered.
6. Turn the `Status` field to `Status []int` to accrue HTTP statuses.
7. Update the `Fprint` function to print errors and statuses in a histogram.
8. Add a logger stage that logs requests when enabled with a "-logger" flag.
9. Use `SplitLimit` instead of `Split` (see pipe.go in the book's repository for details).
10. Write a custom client type that uses the `hit` package's pipeline features to concurrently send requests to "separate" URLs and merge `Results` into a single `Result` using `Result.Merge` (see Section 7.2.6 for an example). You'll discover that `Result` doesn't support "already-merged" `Results` and produces incorrect results. This is because `Result` is designed to work for a single URL.
11. Write an algorithm that automatically calculates the target server's optimal RPS. Enable this algorithm when users don't provide the "-rps" flag.

7.4 Summary

- Concurrency is an implementation detail.
- Concurrency is structuring a program as independently executing components.
- A concurrent pipeline is an extensible and efficient design pattern.
- Design synchronous APIs and leave the concurrency to the caller.

8 Context, HTTP, and Testing

This chapter covers

- Diving into the `net/http` and `httptest` packages.
- Copying a stream of bytes from an `io.Reader` to an `io.Writer`.
- Propagating cancellation signals using the `context` package.
- Implementing Rob Pike's option functions pattern.
- Using `http.RoundTripper` to test the HTTP client.

Previously, we crafted a concurrent pipeline that can simulate HTTP requests to an HTTP server. In this chapter, we'll turn it into a real HTTP client that can actually make requests.

First, we'll explore the `net/http` package to understand how to send HTTP requests. Then, we'll explore the `context` package to learn how to propagate cancelation signals, stop in-flight requests, and still get the results up to that moment. After that, we'll apply the option functions pattern for a streamlined API. Lastly, we'll test the HIT client.

Note

Find the source code of this chapter at the link: https://github.com/inancgumus/gobyexample/tree/main/hit

8.1 The net/http package

Previously, we wrote a function called `Send` to simulate sending a request.

```
func Send(r *http.Request) (Result, error) {
    t := time.Now()

    time.Sleep(100 * time.Millisecond)

    result := Result{        #A
        Duration: time.Since(t),    #A
        Bytes:    10,        #A
        Status:   http.StatusOK,    #A
    }

    return result, nil
}
```

`Send` always returns the same `Result`: the request finishes in 100 milliseconds, 10 bytes are downloaded from the target server, and the request is successful (`http.StatusOK`).

It's time to learn about the `net/http` package to actually send HTTP requests. Once we send an HTTP request with `Send`, we can measure the total number of bytes downloaded from the target server, errors occurred, the HTTP status code, and so on.

8.1.1 Sending requests and reading responses

There are two prominent types that lie in the `net/http` package's heart:

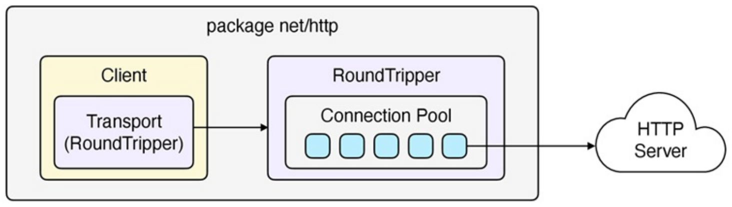
- `Client` to send requests to an HTTP server.
- `Server` to serve HTTP requests to clients.

We'll look into the first one in this chapter and leave the second one for later. The `http.Client` (let's say HTTP client from now on) type is a struct with the following fields.

```
// A Client is an HTTP client.
type Client struct {
    // Transport specifies the mechanism by which individual
    // HTTP requests are made. If nil, DefaultTransport is used.
    Transport RoundTripper

    // CheckRedirect specifies the policy for handling redirects.
    CheckRedirect func(req *Request, via []*Request) error

    // Jar specifies the cookie jar.
    Jar CookieJar

    // Timeout specifies a time limit for requests made by this
    // Client. A Timeout of zero means no timeout.
    Timeout time.Duration
}
```

As Figure 8.1 shows, `Client`, by itself, doesn't establish TCP connections or send requests. Instead, it delegates this responsibility to a `RoundTripper` specified in the `Transport` field.

Figure 8.1 Client uses RoundTripper to send HTTP requests.



As we'll see later, round-trippers can establish TCP connections, send an HTTP request (`http.Request`), and return a response (`http.Response`). Some transporters, such as `http.Transport` can even manage a connection pool to reuse connections for efficiency.

We'll look at the other fields later (except `Jar`, which we won't deal with HTTP cookies).

Note

The default transporter is stored in the `http.DefaultTransport` variable.

Sending a request

We can directly use the default HTTP client stored in the `DefaultClient` variable for convenience when we don't need to customize the client (which we will do so soon).

Once we have a client, we can use its `Do` method to send an HTTP request:

```
// assuming r is *http.Request
// (see the previous code snippet above)
response, err := http.DefaultClient.Do(r)    #A
```

The first result value (`response`) is `*http.Response`, which looks like this:

```
// Response represents the response from an HTTP request.
type Response struct {
    StatusCode int            // e.g. 200
    Body       io.ReadCloser  // response body
    // ..other fields..
}
```

These fields tell us the total number of bytes downloaded and whether the request was successful (e.g., `StatusCode` is 200), which we'll save into a `hit.Result`.

Response.Body

Let's take a closer look at the `Body` field, which is an `io.ReadCloser`:

```
package io

type ReadCloser interface {    #A
    Reader           #B
    Closer           #C
}
```

It's a combination of `Reader` and `Closer` interfaces. So, it has a `Read` and `Close` method.

```
type Reader interface {

    Read(p []byte) (n int, err error)    #A
}
type Closer interface {
    Close() error                #B
}
```

Since `Body` is a `Closer`, we should close it once we consume it. This way, the HTTP client can reuse the connections for the same server to improve performance. For instance:

```
response, err := http.DefaultClient.Do(r)    #A
if err != nil {
    // handle error and return        #B
}
defer response.Body.Close()        #C
// ..process the response body..
```

Here, we're closing the `Body` if there wasn't an error. Otherwise, we don't have to close it as it'll be `nil` if there was an error. This is a common pattern to send an HTTP request in Go.

Note

Like `io.Writer` which standardizes writing a stream of bytes to any resource (such as a file or memory), `io.Reader` standardizes reading a stream of bytes from any resource. See the sidebar "Dive deep: io.Writer and io.Reader" near Section 6.1.2 for more details.

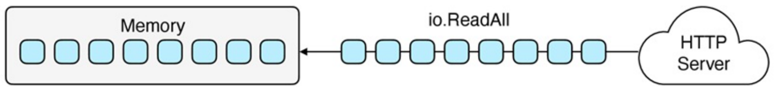
Reading responses

Since `Body` is a `Reader`, it allows us to stream chunks of bytes. Streaming bytes this way can reduce the memory footprint of our programs as we don't need to store the server's complete response in memory. Still, many Go beginners fall into the trap of reading the whole body into memory using `io.ReadAll` (reads from `Reader` to `[]byte`) like this:

```
bytes, err := io.ReadAll(response.Body)    #A
```

Copying bytes this way looks like in Figure 8.2 in memory.

Figure 8.2 Reading and storing everything from the response in memory.

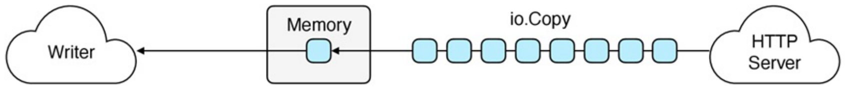


While there are valid use cases for this approach (e.g. if we know the body size), our situation is different: we can't know the body size in advance because we're developing a generic HTTP client that can send requests to any server. We'll take a different approach for efficiency purposes and use the `io.Copy` function instead. It looks like this:

```
func Copy(dst io.Writer, src io.Reader) (written int64, err error
```

`Copy` transfers 32 KB chunks of bytes from a `Reader` to a `Writer`. See Figure 8.3.

Figure 8.3 Reading from the response by reusing a 32 KB buffer in memory.



This means it only uses 32 KB of memory while transferring the bytes, even if the server responds with, say, 1 TB of data. Then, it returns how many bytes are written and an error if there was an error while copying the bytes from the `Reader` to the `Writer`.

Note

`Copy` has a friend called `CopyBuffer`, which allows us to specify the buffer size.

For the HIT client, we only want to get the total number of bytes downloaded from the server. So, we'll use a writer that discards everything written to it: `io.Discard`.

Think of it as `/dev/null` (or `NUL`). This way, `Copy` will read from the server's response body and discard what it reads. Let's look at an example that throws away the bytes being read. This calculates the number of bytes downloaded while fully consuming the response body:

```
numberOfBytesDownloaded, _ := io.Copy(io.Discard, response.Body)
```

This way, we'll be only using 32 KB of memory while reading the entire response body.

To wrap up, the `http` package has a default client (`http.DefaultClient`) that we can use to send HTTP requests to a server. We can use the returned response (`http.Response`) to get the HTTP status code and the response body (`Response.Body`).

We can efficiently read the response body using the `Copy` function with minimal memory overhead. Since `Copy` returns the number of bytes transferred, we can learn about the response body's size and save this in a `Result` to show it in the performance summary.

8.1.2 Wiring up

The next listing shows the `Send` function, which sends an HTTP request to a server. Once the request succeeds, it consumes the response body and returns the performance result.

Listing 8.1 Sending an HTTP request (hit.go)

```
func Send(r *http.Request) (Result, error) {
    t := time.Now()

    var (
        code  int
        bytes int64
    )
    response, err := http.DefaultClient.Do(r)        #A
    if err == nil { // no error                #B
        code = response.StatusCode                #C

        bytes, err = io.Copy(io.Discard, response.Body)    #D

        _ = response.Body.Close()                #E
    }

    result := Result{
        Duration: time.Since(t),
        Bytes:    bytes,
        Status:   code,
        Error:    err,
    }

    return result, err
}
```

We use the default client to send the request.

Then, we stream the response body using `Copy` and throw away the body using `Discard`. Lastly, we close the response body so the default client can reuse the connection. We only do this if the client successfully sends a request to the server. Otherwise, since there wouldn't be a body to read after an error, we neither need to read nor close the body.

`Send` can now send a request to an HTTP server, calculate how many bytes are downloaded, and throw the data away to save memory. We can now test the HIT tool with a favorite web server. For example, we can use httpbin.org:

```
$ go run ./cmd/hit -n 1000 -c 10 http://httpbin.org/get
Sending 1000 requests to "http://httpbin.org/get" (concurrency: 1
Summary:
        Success    : 100%
        RPS        : 53.7
        Requests   : 1000
        Errors     : 0        #A
        Bytes      : 271000    #B
        Duration   : 18.618946s
        Fastest    : 123.959ms
        Slowest    : 1.363453s
```

8.1.3 Optimizing the HTTP client

Once a client establishes a TCP connection, it can send a request to the server. However, establishing a TCP connection is quite chatty and expensive. It's similar to the following:

```
  Server: Hello, would you like to hear a TCP joke? [SYN]
  Client: Yes, I'd like to hear a TCP joke. [SYN,ACK]
  Server: OK, I'll tell you a TCP joke. [ACK]
  Client: OK, I'll hear a TCP joke. [ACK]
  Server: Are you ready to hear a TCP joke? [DATA, ACK]
  Client: Yes, I am ready to hear a TCP joke. [ACK]
  Server: OK, I'm about to send the TCP joke. It will last 10 sec
  Client: OK, I'm ready to hear the TCP joke that will last 10 se
  << HTTP request and response happens somewhere here >>
```

Note

TCP messages are more complex, involving sequence numbers, window sizes, and more.

Luckily, the HTTP protocol allows the server and client to keep the previously established connections alive until the connections time out (it's called *keep-alive*). Clients can reuse opened connections to send requests without establishing new ones.

Let's say the HIT client sends thousands of requests using ten goroutines to the same URL.

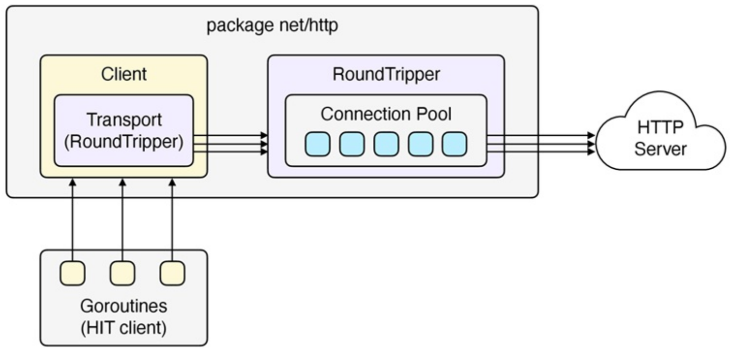
Its performance will be subpar since it currently uses the default HTTP client, which keeps 100 connections open and allows to reuse only two for the same host. The HIT client should reestablish connections each time to send more than two requests to the same host.

While two goroutines can send HTTP requests over two previously established TCP connections, others would need to open new ones to send more. We can't use the default client to measure the full performance of a server. Let's find another way.

Tweaking the connection pool

As Figure 8.4 shows, the HTTP client uses a transporter to establish TCP connections and send HTTP requests to a server. The default HTTP client uses the default transport, which manages a connection pool internally. We should use the same HTTP client when sending requests to take advantage of the connection pool. Otherwise, performance will suffer.

Figure 8.4 HIT client sends requests using the HTTP client's connection pool.



To improve performance, we'll configure a custom client with a custom transport to set the maximum idle connections per host. This will match the HIT client's concurrency level. This way, each goroutine can use a potential idle connection while sending HTTP requests. If the maximum idle connections per host were lower than the concurrency level, the transport would frequently close connections after every request, and performance would suffer.

The next listing shows configuring a custom client and transporter.

Listing 8.2 Customizing Client configuration (client.go)

```
func (c *Client) do(r *http.Request, n int) <-chan Result {
    pipe := produce(n, func() *http.Request {
        return r.Clone(context.TODO())
    })

    if c.RPS > 0 {
        t := time.Second/time.Duration(c.RPS*c.concurrency())
        pipe = throttle(pipe, t)
    }

    client := c.client()                        #A
    defer client.CloseIdleConnections()               #B

    return split(pipe, c.concurrency(), func(r *http.Request) Res
        // the final result will handle the error.
        result, _ := Send(client, r)
        return result
    })
}

func (c *Client) client() *http.Client {
    return &http.Client{                       #D
        Transport: &http.Transport{               #E
            MaxIdleConnsPerHost: c.concurrency(),          #F
        },
        CheckRedirect: func(_ *http.Request, _ []*http.Request) e
            return http.ErrUseLastResponse               #G
        },
    }
}
```

We've created a custom client with a tuned transport to improve performance.

There is a new method called `client`, which configures a new HTTP client with a custom transport to improve performance. Since we want to take advantage of the connection pooling, we call `client` once before running the splitter stage. Then, inside the closure (`split`'s third parameter), we share the same returned HTTP client with goroutines.

Another optimization we made is using the `CloseIdleConnections` method.

Using `CloseIdleConnections`, we forcefully close idle connections when `do` returns. This reduces memory usage and other operating system resources, such as file descriptors.

Lastly, we're setting `CheckRedirect` to prevent redirects (where one request redirects to another). This is crucial when testing redirecting URLs, and that might skew performance results. Otherwise, we might inadvertently measure the combined response times of the initial and redirected URLs, leading to misleading data about the server's performance.

Customizing the client and transporter

The `Send` function will use our tuned client to send requests. See the next listing. We use a performance-tuned client instead of the default client (unlike previously).

Listing 8.3 Accepting a custom client (hit.go)

```
func Send(c *http.Client, r *http.Request) (Result, error) {    #

    t := time.Now()

    var (
        code  int
        bytes int64
    )
    response, err := c.Do(r)                #A
    if err == nil { // no error
        code = response.StatusCode
        bytes, err = io.Copy(io.Discard, response.Body)
        _ = response.Body.Close()
    }

    result := Result{
        Duration: time.Since(t),
        Bytes:    bytes,
        Status:   code,
        Error:    err,
    }

    return result, err
}
```

Let's benchmark the default and the performance-tuned client.

Before:

```
$ go run ./cmd/hit -n 1000 -c 10 http://localhost:8080
    RPS         : 2200
    Duration    : 46s
```

After:

```
$ go run ./cmd/hit -n 1000 -c 10 http://localhost:8080
    RPS         : 22000
    Duration    : 4.5s
```

Note

I'm running these against an unloaded local HTTP server to see accurate results.

There is a 10X difference. Previously, we didn't effectively use the connection pool. This caused connection congestion and prevented goroutines from being performant. Later on, we've performance-tuned the client with a custom transport. This provided goroutines reestablished connections from the connection pool to be more performant.

With these changes, the HIT client became more performant. It uses the same HTTP client to exploit the connection pool when sending HTTP requests and avoids unnecessarily re-establishing TCP connections to the same host. The pool reacquires the connection when the request ends. It then releases the connection for use by another request.

Timing out requests

Imagine a goroutine takes a request from the HIT client's concurrent pipeline and sends a request, which takes a long time. That goroutine can't pick the next request from the pipeline until it finishes its work. Once other goroutines also have the same issue simultaneously, the HIT client might take forever to return performance results.

However, if we define a *timeout per request*, goroutines can report an error and continue when a specified duration is surpassed instead of clogging the pipeline.

See the next listing.

Listing 8.4 Setting a timeout per request (client.go)

```
type Client struct {
    C       int           // C is the concurrency level
    RPS     int           // RPS throttles the requests per secon
    Timeout time.Duration // Timeout per request
}

func (c *Client) client() *http.Client {
    return &http.Client{
        Timeout: c.Timeout, // zero means no timeout
        Transport: &http.Transport{
            MaxIdleConnsPerHost: c.concurrency(),
        },
        CheckRedirect: func(_ *http.Request, _ []*http.Request) e
            return http.ErrUseLastResponse
        },
    }
}
```

We've added a new `Timeout` field to the `Client` type and set the HTTP client's `Timeout` field. We can now set timeout per request using the new `Timeout` field:

```
client := &hit.Client{
    ...
    Timeout: 30 * time.Second,
}
```

When we don't set it, the HTTP client's `Timeout` field will be zero, meaning requests won't timeout. We can also set a sensible default if this field isn't set. This way, the package's usage will be more convenient. Let's leave this as an exercise for you.

8.1.4 Wrap up

- `http.Client` allows us to send HTTP requests.
- `Client.Do` sends an HTTP request.
- `http.Response` represents the server's response.
- `Response.Body` provides a way to read the server's response.
- `io.Copy` efficiently reads from a `Reader` and writes to a `Writer`.
- `io.Discard` is an `io.Writer` that throws away data without writing it.
- `http.Transport` establishes TCP connections, sends HTTP requests, and pools connections. Its `MaxIdleConnsPerHost` field controls the number of idle connections to keep in the connection pool per host.
- `Client.CloseIdleConnections` forcefully closes idle connections.
- `Client.Timeout` controls the timeout threshold per request.

The HTTP client has other options to customize its behavior. See them all at the links https://pkg.go.dev/net/http#Client and https://pkg.go.dev/net/http#Transport.

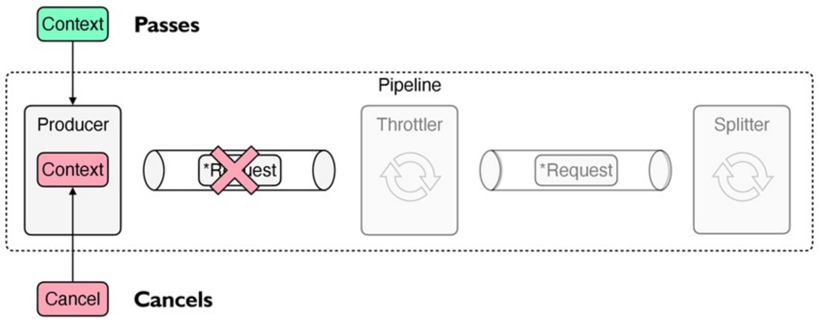
8.2 Context: Propagating cancelation

Imagine we want to send millions of requests and, for some reason, want to cancel the requests. Or, we might want to stop sending requests after a specific time (timeout). However, the client's pipeline once started keeps going on until it sends all requests.

How can we stop the pipeline at will?

The idiomatic way to do cancellation is to use the `context` package's `Context` type, which propagates cancellation signals. By using `Context`, we can stop ongoing work across packages (including goroutines). See Figure 8.5 for an illustration.

Figure 8.5 The pipeline stops once the passed context is canceled.



We pass a `Context` to the pipeline. Once we cancel the `Context`, the producer detects that it is canceled, closes its output channel, and returns, causing a domino effect in the pipeline. Following this, other stages automatically stop as they depend on the producer's channel.

We'll first dive into the `context` package. Then, we'll make the producer accept a `Context` (stopping once the `Context` is canceled). Lastly, we'll pass `Context` from the HIT tool to the HIT client. This will allow the HIT tool to stop the HIT client when needed (e.g., after a timeout, when users terminate the program while it's running (e.g., pressing CTRL+C).

8.2.1 The context.Context type

This is what the `context` package's `Context` type looks like:

```
package context

type Context interface {
    Done() <-chan struct{}
    Err() error
    // ...
}
```

We see two methods (which we'll look at how to use them soon):

- `Done` returns a channel to check if the `Context` is canceled.
- `Err` returns a non-nil error if the context is canceled.

Let's first discuss the mechanics of using a `Context`. Then, we can dig into these methods.

Note

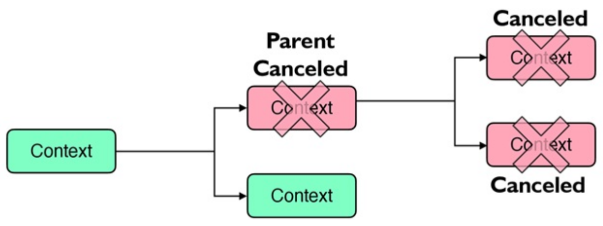
For formatting purposes, we might call `Context` "context" from now on.

What is the context?

Think of contexts like a tree with roots and leaves (or parent-child). See Figure 8.6.

The most crucial fact about contexts is that they are hierarchic. When one of the parent contexts cancels, all the child contexts derived from that parent are also canceled. For instance, in the figure, we see five contexts. Those on the left are the parent contexts of some of those on the right. Once the middle-top context is canceled, it cancels all of its children (the ones on the most right since they are derived from that parent context).

Figure 8.6 Canceling the parent cancels its children. The cross-out indicates the cancellation in the figure. Cancellation signals propagate through the hierarchy.



Since contexts are hierarchical, we need a root context to begin with. We can create a root context either by using the `Background` or `TODO` function. For instance:

```
root := context.Background()    #A
```

This is a root context and it cannot be canceled. Its unexported underlying concrete type is hidden since all context functions return the `Context` interface. This allows us to chain contexts. For example, we can *derive* a timeout context from the root context like this:

```
child, cancel := context.WithTimeout(root, 15*time.Second)    #A
```

This context is cancelable and will be automatically canceled (it runs an internal goroutine to track this) after fifteen seconds if we don't cancel it earlier. We can cancel it before its time by calling the returned `cancel` function.

Note

We should always call the `cancel` function to release the acquired resources after we finish working with the context (whether we want to cancel the context or not).

We can further chain the previous context if we want, building a tree of contexts:

```
grandChild, cancel := context.WithTimeout(child, 5*time.Second)
```

This one has a lifetime of five seconds before it gets canceled. Notice that it cannot surpass its parent context's lifetime, which is fifteen seconds. This is because the cancellation in a parent context also cancels its children. We might create a tree of context when we want granular control. For example, we can give one function the time to finish its job in fifteen seconds, and then that function can call another function, giving that function five seconds.

For example:

```
func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 15*t
    defer cancel()

    shouldFinishIn15Seconds(ctx)

    // ...
}

func shouldFinishIn15Seconds(ctx context.Context) {
    ctx, cancel := context.WithTimeout(ctx, 5*time.Second)
    defer cancel()

    go shouldFinishIn5Second(ctx)

    // ...
}
```

Here, we want the first function to finish its work in fifteen seconds. Once the function surpasses this time, the context (the one in `main`) will be automatically canceled. That function derives another context and wants the next function to finish its work in five seconds. This last context will be canceled before its parent context. Canceling this grandchild context won't affect the parent context.

Context communication is one way: only from parents to children. Children cannot cancel their parent contexts (and should not). I was once involved in a project like that, which didn't go well. It wasn't easy to understand and track which context canceled another. It has taken us weeks to figure out the underlying bug. Then, we removed this wild custom context (yes, we can create custom contexts because the `Context` type is an interface).

Another fact about contexts is that they are immutable. That's why we derive new contexts instead of changing the existing ones. For instance, we can't modify a timeout context's deadline once it's created. Being immutable makes it easy to reason about context lifecycles and safely propagate them into goroutines.

Propagating cancelation signals

We said that context propagates cancellation signals.

We can learn if a context is canceled by calling `Done`. Since `Done` returns a channel, we can listen to this channel until the context is canceled. For instance, we can use a timeout context to stop waiting for a long-running operation if that operation takes too long.

```
ctx, cancel := context.WithTimeout(ctx, time.Minute)
defer cancel(

select {
case err := <-runLongRunningOperation():        #A
    return err
case <-ctx.Done():                #B
    // context is timed out
    // or, we called cancel() to cancel the context
    // or, one of the parent contexts is canceled somehow
    return ctx.Err()                #C
}
```

This code listens to two channels at once. If the long-running operation does not send a message to its channel in a minute, the second case is selected, and we stop waiting. Keep in mind that even if we stop waiting, the long-running operation might still continue.

The best way to stop that operation is to pass a context and check if it's canceled:

```
func runLongRunningOperation(ctx context.Context) {    #A

    for {
        // ...do interesting stuff...

        if err := ctx.Err(); err != nil {        #B
            return err
        }
    }
}
```

We'll see more examples of the `context` package throughout the book. Now that we've looked at the basics, let's apply this knowledge to the HIT client.

Stopping the producer stage

As the next listing shows, we're making the producer stage context-aware. It listens to the context's `Done` channel to track whether the context is canceled and returns if so.

Listing 8.5 Stopping the producer (pipe.go)

```
package hit

import (
    "context"
    ...
)

// Produce calls fn n times and sends results to out.
//
// fn should return when ctx is canceled.
func Produce(
    ctx context.Context,             #A
    out chan<- *http.Request, n int, fn func() *http.Request,
) {
    for range n {
        select {               #B
        case <-ctx.Done():        #C
            return            #C
        case out <- fn():            #D
        }
    }
}

func produce(
    ctx context.Context,
    n   int, fn func() *http.Request,
) <-chan *http.Request {
    out := make(chan *http.Request)
    go func() {
        defer close(out)
        Produce(ctx, out, n, fn)        #E
    }()
    return out
}
```

Tip

A function that accepts a context should always check if the context is canceled. Also, it's idiomatic to declare and pass the context as the first function parameter.

We've modified the producer to accept a context.

Recall that the producer sends requests to its output channel for other stages to consume. When this context is canceled, `select` picks the `Done` case and causes the producer to close its output channel. This creates a chain reaction in the pipeline that stops every stage.

We can now stop the producer when we want. Since the producer takes a context (an interface), this approach gives callers more options for stopping the producer. They can pass any context implementation (e.g., a timeout context).

However, there is one issue with this function. `fn` in `Produce` might not stop even after the context has been canceled. So, we added a warning as a comment and trust the caller will do the right thing: "*fn should return when ctx is canceled*." When calling `Produce`, the user has the context (`context.Context`) and is responsible for ensuring `fn` stops accordingly.

This approach leans on the pragmatic nature of Go, trusting that programmers are responsible and will consult the documentation before using a package's functionality.

Note

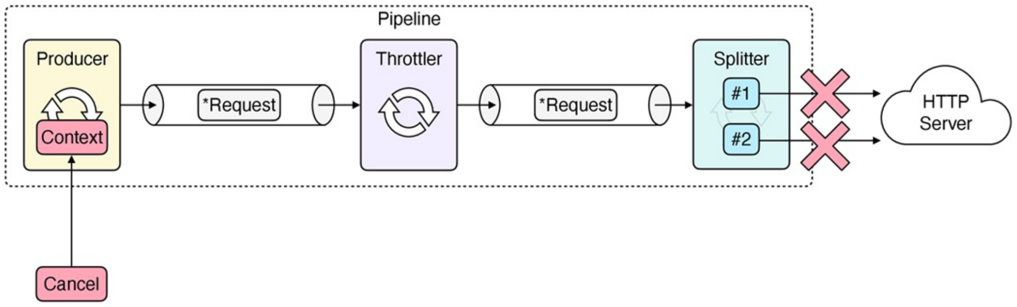
Channels can be ready simultaneously, and the select statement may pick the second channel even after the context is canceled. This might cause the producer to produce another request. We should check the context cancellation in the `fn` function if we want certainty.

Canceling in-flight requests

When a context is canceled, the producer now stops producing requests, but what about the requests that are already in progress? The splitter may have already received the request values from the producer (*or the throttler*) and started sending requests to a server.

Can we stop these in-flight requests, too? Yes! See Figure 8.7.

Figure 8.7 Canceling the parent context, cancels the in-flight requests.



The `http` package allows us to stop in-flight requests. We only need to set the request's context (`*http.Request`) to a cancelable context while cloning the request (e.g., `context.WithTimeout`). This way, we can stop in-flight requests. See the next listing.

Listing 8.6 Canceling in-flight requests (client.go)

```
func (c *Client) Do(ctx context.Context, r *http.Request, n int)

    t := time.Now()

    var sum Result
    for result := range c.do(ctx, r, n) {                    #B
        sum = sum.Merge(result)
    }

    return sum.Finalize(time.Since(t))
}

func (c *Client) do(ctx context.Context, r *http.Request, n int)
    pipe := produce(ctx, n, func() *http.Request {
        return r.Clone(ctx)                              #C
    })

    if c.RPS > 0 {
        t := time.Second/time.Duration(c.RPS*c.concurrency())
        pipe = throttle(pipe, t)
    }

    client := c.client()
    defer client.CloseIdleConnections()

    return split(pipe, c.concurrency(), func(r *http.Request) Res
        result, _ := Send(client, r)
        return result
    })
}
```

We pass the incoming context while cloning the request. This way, the `http` package can stop the in-flight requests when the context is canceled.

Passing a context to the HIT client

Let's give our freshly modified context-aware functions a try.

We might want to stop HTTP requests after a certain time, let's say after ten minutes.

So, we'll set a shorter timeout to see the effects of cancelation. Since the HIT tool runs the HIT client, it's a good place to create a root context (`Background` and `TODO` return root contexts, which are non-cancelable by their nature). See the next listing.

Listing 8.7 Canceling in-flight requests (client.go)

```
package main

import (
    ...
    "context"
)

func runHit(e *env, c *config) error {
    handleErr := func(err error) error {
        if err != nil {
            fmt.Fprintf(e.stderr, "\nerror occurred: %v\n", err)
            return err
        }
        return nil
    }

    // exercise: define timeout flags and use them here
    const (
        timeout           = 5 * time.Second
        timeoutPerRequest = 30 * time.Second  // we'll use this l
    )
    ctx, cancel := context.WithTimeout(context.Background(), time
    defer cancel()                            #B

    request, err := http.NewRequest(http.MethodGet, c.url, http.N
    if err != nil {
        return handleErr(fmt.Errorf("new request: %w", err))
    }

    client := &hit.Client{
        C:   c.c,
        RPS: c.rps,
    }
    sum := client.Do(ctx, request, c.n)                   #C
    sum.Fprint(e.stdout)

    if err = ctx.Err(); errors.Is(err, context.DeadlineExceeded)
        return handleErr(fmt.Errorf("timed out in %s", timeout))
    }
```

```
                                                 #D
    return handleErr(err)                                           #E
}
```

Remember

Context is like a tree structure, starting with a root context. Each leaf can have contexts with different policies and timeouts. Cutting (canceling) the root kills the whole tree.

Using `WithTimeout`, we're creating a timeout context and passing it to the HIT client.

```
ctx, cancel := context.WithTimeout(context.Background(), timeout)

...
sum := client.Do(ctx, request, c.n)
```

This context will cancel itself after one second or when we call the `cancel` function.

We're also checking `Err` and returning a user-friendly error if the context is canceled due to timeout. `Err` returns `DeadlineExceeded` if the context is timed out.

```
if err = ctx.Err(); errors.Is(err, context.DeadlineExceeded) {

    return handleErr(fmt.Errorf("timed out in %s", timeout))
}
```

This way, we can avoid printing non-user friendly *"context deadline exceeded"* error messages. Instead, we'll print "*timed out in 5s*" if the context was canceled. Another error we could get is the `context.Canceled` error, which happens if the context is canceled for an unknown reason. We could also customize this error similarly if we want to do so.

To wrap up, we've propagated a timeout context to the HIT client, which propagates it downstream. For instance, ongoing requests will stop once the context is canceled (*after five seconds*). We've also customized the timeout error with a user-friendly message.

Trying it out

Let's give it a try.

```
$ go run ./cmd/hit -n 1000 -c 10 http://example.com
 __  __      __      ______
/\ \_\ \   /\ \   /\__  _\
\ \  __ \  \ \ \  \/_/\ \/
 \ \_\ \_\  \ \_\    \ \_\
  \/_/\/_/   \/_/     \/_/

Sending 1000 requests to "http://example.com" (concurrency: 10)

Summary:
        Success    : 86%
        RPS        : 69.9
        Requests   : 70
        Errors     : 10
        Bytes      : 75260
        Duration   : 1.001584s
        Fastest    : 27.214ms
        Slowest    : 393.778ms

error occurred: timed out in 5s
```

Although we wanted to send one thousand requests, the client sent seventy due to timeout.

Dive deep: context.WithCancelCause and context.Cause

`WithCancelCause` function allows us to set the cancellation reason for a context.

It returns a cancel function that we can use to pass an error.

```
ctx, cancel := context.WithCancelCause(parentContext)

// ...
cancel(errors.New("higgs boson doomsday"))
// ...
err := context.Cause(ctx)
fmt.Println(err) // prints "higgs boson doomsday"
```

This is useful, especially in programs where the cancellation reasons are important to debug or log the issues to understand what's going on. Visit the link to read more about the proposal to see why we have this function: https://github.com/golang/go/issues/51365

8.2.2 The signal package: Is CTRL+C the end?

We won't see the performance results if we press CTRL+C while the HIT tool runs. The operating system sends an interrupt signal that causes our program to stop immediately:

```
$ go run ./cmd/hit -n 1000 -c 10 http://example.com
 __  __     __     ______
/\ \_\ \   /\ \   /\__  _\
\ \  __ \  \ \ \  \/_/\ \/
 \ \_\ \_\  \ \_\    \ \_\
  \/_/\/_/   \/_/     \/_/

Sending 1000 requests to "http://example.com" (concurrency: 10)
^C signal: interrupt
$            #A
```

The tool might have run many requests, but we can't see the results.

Luckily, we can capture this signal to show the results before the tool shuts down. This will provide a better user experience since users won't miss the result accrued until that point.

For that, we can use the `signal` package's `NotifyContext` function:

```
package signal

func NotifyContext(
    parent  context.Context,    #A
    signals ...os.Signal,         #B
) (
    ctx  context.Context,        #C
    stop context.CancelFunc,    #D
)
```

Essentially, as with other typical context functions, `NotifyContext` derives a new context from another context as its parent. However, it's more interesting than meets the eye.

Until we call the returned `stop` function, CTRL+C (for example) won't be able to terminate the program. So, it's critical to defer this `stop` function. Otherwise, we can't exit the program using CTRL+C if it never returns (unless we kill the program's process!).

As the next listing shows, we're using `NotifyContext` to track the interrupt signal.

Listing 8.8 Catching interruptions (cmd/hit/hit.go)

```
package main

import (
    ...
    "os/signal"
)

func runHit(e *env, c *config) error {
    ...

    const (
        timeout           = time.Hour
        timeoutPerRequest = 30 * time.Second
    )

    ctx, cancel := context.WithTimeout(context.Background(), time
    ctx, stop := signal.NotifyContext(ctx, os.Interrupt)
    defer cancel()                                   #B
    defer stop()                                        #C

    ...
}
```

We derived a new notification context from the timeout context. Because the timeout context is the notification context's parent, when the timeout occurs, the notification context will be canceled. So, we bumped the timeout duration to see the notification context in action.

```
const (
    timeout = time.Hour
    // ...
)
```

Let's now run the program and immediately press CTRL+C .

```
$ go run ./cmd/hit -n 1000 -c 10 http://example.com
 __  __     __     ______
/\ \_\ \   /\ \   /\__  _\
\ \  __ \  \ \ \  \/_/\ \/
 \ \_\ \_\  \ \_\    \ \_\
  \/_/\/_/   \/_/     \/_/

Sending 1000 requests to "http://example.com" (concurrency: 10)
^C                   #A
Summary:
        Success    : 88%
        RPS        : 69.7
        Requests   : 80
        Errors     : 10
        Bytes      : 87920
        Duration   : 1.146961s
        Fastest    : 19.569ms
        Slowest    : 292.765ms

error occurred: context canceled
exit status 1
```

Pressing CTRL+C causes `NotifyContext` to capture the interrupt signal, preventing exiting the program. Then, the notification context is canceled. This signal is propagated to the HIT client because all the components in the client are waiting for a context cancellation signal.

Once the HIT client stops, it returns the performance results up to the moment the context is canceled. We see the summary even after the program is interrupted. This provides a better user experience because users can now see the summary after terminating the program.

8.2.3 Wrap up

- The `context` package propagates cancellation signals.
- `context.WithTimeout` returns a `context.Context` that cancels after a timeout.
- `Context.Err` returns the `context.Context`'s cancellation reason. The returned error is `context.DeadlineExceeded` if the context is timed out, `context.Canceled` if the context is canceled for another reason, or `nil`

if it's not yet canceled.
- `signal.NotifyContext` returns a `context.Context` that automatically cancels after receiving the specified signal from the operating system.

8.3 Option functions pattern

In this section, we'll build upon the existing low-level HIT client API, utilizing the existing primitives it provides, and paint it with a higher-level, slightly more convenient-to-use API.

This higher-level API will wrap around the low-level HIT client API. API users can still build their clients, pipelines, and abstractions using the low-level API, but the new higher-level API will be for situations where we might not want to configure the low-level API.

Note

Think of the new API as a Façade, which masks a more complex underlying API.

We'll provide the new API without changing any code in the low-level API.

Providing a convenience helper

The `hit` package requires a few steps to set up. For instance:

```
request, err := http.NewRequest(http.MethodGet, url, http.NoBody)
if err != nil {
    return err
}
var client hit.Client
sum := client.Do(ctx, request, 1_000_000)
```

It might be a better user experience if the package allowed sending requests like this:

```
sum, err := hit.SendN(ctx, url, 1_000_000)
// check the website's performance using the sum variable.
```

Straightforward to use! `SendN` is a *wrapper* around `Client.Do`, making it easy to send requests. The next listing shows `SendN`'s implementation.

Listing 8.9 Implementing the SendN function (hit.go)

```
// SendN sends n HTTP requests to the url and returns an aggregat
func SendN(ctx context.Context, url string, n int) (Result, error
    r, err := http.NewRequest(http.MethodGet, url, http.NoBody)
    if err != nil {
        return Result{}, fmt.Errorf("new http request: %w", err)
    }

    var c Client

    return c.Do(ctx, r, n), nil
}
```

The `hit` package is now easier to use thanks to `hit.SendN`. Users can use `hit.Client` if they want to customize the rest of the options, like the concurrency level and timeout.

Providing optional arguments

```
Users must use hit.Client if they want to customize the client. F
func SendN(
  ctx context.Context,
  url string,
  n,
  c int,
  rps int,
  timeout time.Duration,
) (Result, error)
```

This complicates the API and makes it cumbersome to use it. While we were trying to provide a better user experience, we might be making it worse. We're back at square one.

Moreover, the first three parameters are mandatory, but the rest are not. How can we indicate that some are optional? In his blog, one of the creators of Go, Rob Pike shares a pattern called *Option Functions* to solve this problem. The idea is to pass an arbitrary number (*variadic*) of functions as options to customize the behavior of an API.

Note

Find more detail about the option functions pattern at the link https://commandcenter.blogspot.com/2014/01/self-referential-functions-and-design.html

For instance, `SendN` would look like this when we apply this pattern:

```
func SendN(
    ctx context.Context, url string, n int, opts ...Option,
) (Result, error) {
```

Note

: A variadic function accepts an arbitrary number of input values—zero or more. The ellipsis (three-dots) prefix before an input type makes a function variadic.

Then, we can call the function like so:

```
hit.SendN(
    context.Background(),
    "http://somewhere",
    1_000,
    hit.Concurrency(10),
    hit.RequestsPerSecond(1),
    hit.Timeout(time.Second),
)
```

This sets the HIT client's `C`, `RPS`, and `Timeout` fields. Since the function is variadic, users don't have to provide every option. For instance, they can just set the concurrency level:

```
hit.SendN(
    context.Background(),
    "http://somewhere",
    1_000,
    hit.Concurrency(10),
)
```

Or, they might provide no options at all (this preserves the original API's behavior):

```
hit.SendN(context.Background(), "http://somewhere", 1_000)
```

This last one will use the defaults for the rest of the options, such as the concurrency level.

What are option functions?

Now that we looked at the benefits of Rob Pike's option functions pattern, let's look at how to implement it. Each option function is a closure to change a field of `hit.Client`. For instance:

```
func Concurrency(n int) func(c *Client) {    #A
    return func(c *Client) { c.C = n }    #B
}
```

We can make this function's signature more readable by defining a new `Option` type:

```
type Option func(c *Client)        #A


func Concurrency(n int) Option {
    return func(c *Client) { c.C = n }
}
```

The `Concurrency` option returns a closure that sets the HIT client's `C` field. For instance, passing 10 to `Concurrency` returns a closure that sets the `C` field to 10 *once called*:

```
var client Client    // C is 0
c := Concurrency(10) // C is still 0
c(&client)           // C is 10
```

It's a form of lazy initialization. Calling the returned closure does the actual job.

The remaining options look like this:

```
func RequestsPerSecond(d int) Option {

    return func(c *Client) { c.RPS = d }
}
```

```
func Timeout(d time.Duration) Option {
    return func(c *Client) { c.Timeout = d }
}
```

As a side benefit, since the option functions return the same type, Go would automatically group each option function under the `Option` type in the package documentation:

type Option

    func Concurrency(n int) Option
    func RequestsPerSecond(d int) Option
    func Timeout(d time.Duration) Option

Since `SendN` takes a variadic `Option` function, users can provide any of these, as we've seen. Inside the `SendN` function, we'll call the provided ones to let them set the `Client`'s fields:

```
func SendN(ctx context.Context, url string, n int, opts ...Option

    // ..
    var c Client
    for _, o := range opts {
        o(&c)
    }
    return c.Do(ctx, r, n), nil
}
```

Here, the function runs each closure returned from function options, letting them set the `Client`'s fields. Lastly, `SendN` calls `Client`'s `Do` method to send requests.

Implementing the option functions pattern

Now that we looked at the option functions pattern, let's restructure our client.

See the next listing.

Listing 8.10 Implementing the option functions (hit.go)

```
// Option allows changing Client's behavior.
type Option func(*Client)

// Concurrency changes the Client's concurrency level.
func Concurrency(n int) Option {
    return func(c *Client) { c.C = n }
}

// RequestsPerSecond changes Client's RPS (requests per second).
func RequestsPerSecond(d int) Option {
    return func(c *Client) { c.RPS = d }
}

// Timeout changes the Client's timeout per request.
func Timeout(d time.Duration) Option {
    return func(c *Client) { c.Timeout = d }
}

// SendN sends n HTTP requests to the url and returns an aggregat
func SendN(ctx context.Context, url string, n int, opts ...Option
    r, err := http.NewRequest(http.MethodGet, url, http.NoBody)
    if err != nil {
        return Result{}, fmt.Errorf("new http request: %w", err)
    }

    var c Client
    for _, o := range opts {
        o(&c)
    }

    return c.Do(ctx, r, n), nil
}
```

We declare an `Option` type to group all the options and make the code more readable and concise. This type also makes it consistent to see which functions are options. Later on, we declare the function options, like `Concurrency`, `RequestsPerSecond`, and `Timeout`.

Then, we're changing `SendN`'s signature to take a variadic option function. Lastly, we're calling each option function's returned closure to let them set the client's fields.

Note

We could set additional default values for the new client before overwriting the client fields with the option functions. However, I don't suggest doing it there. Instead, we can follow the way we did with the concurrency function in Listing 6.11 earlier. Doing so lets users get the same default behavior when they create a new `Client` or use the `SendN` function.

There are three options; each returns a closure to set the HIT client's fields. Users can now pass the options to the `SendN` function to customize the sending of requests.

Wrap up

We now have a higher-level convenience function (`SendN`) to do everything the `hit` package's low-level `Client` can. The `Client` type still provides low-level functionality and it remains to be composable like LEGO bricks, allowing users to build their custom client implementations.

```
Although the function options pattern can make a streamlined API,
type Options struct {
    C       int
    RPS     int
    Timeout time.Duration
}
```

```
func SendN(ctx context.Context, url string, n int, opts Options)
```

It's a matter of personal preference. Both solutions are fine.

8.4 Testing

We have two options to test the hit HTTP client:

- Launch a test server using `httptest.Server`.
- Intercept requests with `http.RoundTripper`.

Let's start with the first one.

8.4.1 Testing with a test server

Before getting started, we should clarify a few concepts.

- Underneath, a test server is an `http.Server` tuned for testing purposes.
- An `http.Server` listens for client connections and serves requests with a handler.

We can use `httptest.NewServer` to spawn a new test server:

```
package httptest

func NewServer(handler http.Handler) *httptest.Server
```

A handler can be an ordinary function with the following signature.

```
ordinaryFunc := func(_ http.ResponseWriter, _ *http.Request) {

    /* code to serve an HTTP request */
}
```

We should convert this function to an `http.Handler` to register on `httptest.Server`. For that, we can use the `http.HandlerFunc` type. Here's an example:

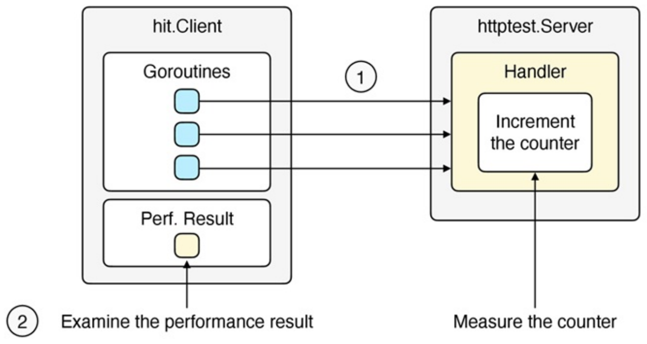
```
handler := http.HandlerFunc(ordinaryFunc)    #A
server  := httptest.NewServer(handler)    #B
defer server.Close()
```

We've converted `handler` to an `http.Handler` that we can register on the test server. The test server will call this handler whenever the server receives an HTTP request. Chapter 9 dives into servers and handlers in greater detail. For now, we're ready to test the HIT client.

Testing the client

See Figure 8.8 for our plan to test the HIT client. We send requests to the test server, and check whether the counter inside the HTTP request handler incremented at the same rate as the number of requests we send to the server. We also examine the performance result.

Figure 8.8 Sending requests to the test server and measuring the results.



The next listing brings our plan to life. We're launching a test server, incrementing the counter on each HTTP request, and then measuring the total number of hits and errors.

Listing 8.11 Testing the client with SendN (hit_test.go)

```
package hit

import (
    "context"
    "net/http"
    "net/http/httptest"
    "sync/atomic"
    "testing"
)

func TestSendN(t *testing.T) {
    t.Parallel()

    var gotHits atomic.Int64                #A
    server := httptest.NewServer(http.HandlerFunc(         #B
        func(_ http.ResponseWriter, _ *http.Request) {
            gotHits.Add(1)                  #C
        },
    ))
    defer server.Close()                    #D

    const wantHits, wantErrors = 10, 0

    sum, err := SendN(                      #E
        context.Background(), server.URL, wantHits,        #E
    )                              #E
    if err != nil {
        t.Fatalf("SendN() err=%q; want nil", err)
    }
    if got := gotHits.Load(); got != wantHits {        #F
```

```
        t.Errorf("hits=%d; want %d", got, wantHits)
    }
    if got := sum.Errors; got != wantErrors {        #G
        t.Errorf("Errors=%d; want %d", got, wantErrors)
    }
}
```

Let's go over this code.

We're first declaring an atomic 64-bit integer counter (`gotHits`). This is necessary because the handlers are run concurrently. An atomic integer prevents data race issues.

```
var gotHits atomic.Int64
```

Then, we're launching a new test server with a handler that increments the counter. This way, we can measure how many times the HIT client sends a request to this handler. Once we launch the test server, using `defer`, we close it right before the test function returns.

```
server := httptest.NewServer(...)
defer server.Close()
```

Lastly, we're loading the counter (`gotHits.Load()`) to measure the number of requests. We're also checking the number of errors occurring while sending requests. Let's try it.

```
$ go test ./hit -v
```

```
--- PASS: TestSendN
```

The HIT client made ten requests to the test server without errors.

Note

For more information about atomic types, visit https://pkg.go.dev/sync/atomic.

8.4.2 Testing with a round-tripper

Now that we have tested the HIT client with a test server, let's try another approach. We'll intercept the requests the HIT client makes with a fake `http.RoundTripper` implementation.

`RoundTripper` is an interface with a single `RoundTrip` method, which takes a `Request` to receive request details, like the request URL, and returns a `Response`.

```
type RoundTripper interface {

    RoundTrip(*http.Request) (*http.Response, error)
}
```

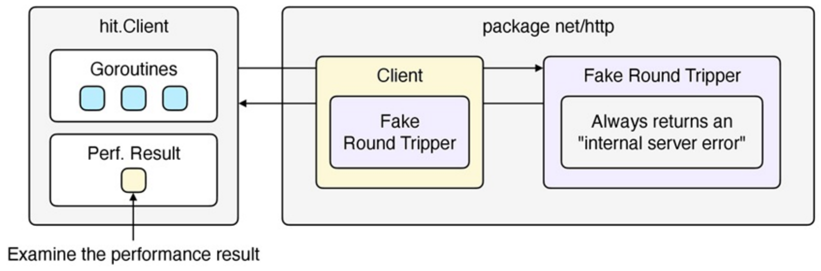
In *Section 8.2.3 Optimizing the HTTP client*, we discussed that `http.Client` has a `Transport` field to establish TCP connections and send HTTP requests. This field's type is `RoundTripper`.

```
package net/http

type Client struct {
    // Transport specifies the mechanism by which individual
    // HTTP requests are made.
    // If nil, DefaultTransport is used.
    Transport RoundTripper

    // ..other fields..
}
```

So, we can inject a fake `RoundTripper` in `hit.Client` to intercept HTTP requests. To do that, we should add a `Transport` field to `hit.Client` to allow customizing the transport mechanism it uses. Then, we can craft and test the client with a fake `RoundTripper`.

See Figure 8.9 for an illustration.

Figure 8.9 The fake round tripper always returns an internal server error.



Without knowing, the HIT client sends requests using an HTTP client with a fake round-tripper. This fake tipper always returns an internal server error. Once the HIT client returns a performance result, the `Errors` field should match the number of requests sent (because every request will end up in an error due to the internal server error).

Note

We won't send requests to any server because we'll be using a fake round-tripper. The fake tripper would simulate sending requests and returning responses (that's what round-trippers primarily do: sending requests and returning responses).

Adding a Transport field

As the next listing shows, we're adding a `Transport` field to `hit.Client`.

Listing 8.12 Adding a Transport field (client.go)

```
type Client struct {
    C         int               // C is the concurrency level (de
    RPS       int               // RPS throttles the requests per
    Timeout   time.Duration     // Timeout per request
    Transport http.RoundTripper // Transport to use for the reque
}

func (c *Client) client() *http.Client {
    transport := c.Transport
    if transport == nil {
        transport = &http.Transport{
            MaxIdleConnsPerHost: c.concurrency(),
        }
    }
    return &http.Client{
        Timeout: c.Timeout,
        CheckRedirect: func(_ *http.Request, _ []*http.Request) e
            return http.ErrUseLastResponse
        },
        Transport: transport,
    }
}
```

We can either set this field to a custom transport we prepare or leave it to a test to overwrite it. The previous one (custom transport) is exactly what we're doing in the `client` method right now. We're setting `Transport` to a custom transport if it's not set. Otherwise, the client will directly use the `Transport` field. This allows tests to overwrite the transport behavior.

We've added a `Transport` field to allow tests to decide how the HIT client makes requests. We can now set this field from tests to intercept the HIT client's requests.

Crafting a fake tripper

Now that the HIT client allows modifying its `Transport`, we can craft a fake.

See the next listing.

Listing 8.13 Crafting a fake transport (client_test.go)

```
package hit

import "net/http"

type fakeTripper func(*http.Request) (*http.Response, error)

func (f fakeTripper) RoundTrip(r *http.Request) (*http.Response,
    return f(r)
}
```

We're declaring a function type with the same signature of the `RoundTripper` interface's `RoundTrip` method. Then, we're adding a `RoundTrip` method to `fakeTripper` that calls the current function that we will convert to the `fakeTripper` type.

Using the fake tripper

We've crafted a fake `RoundTripper` that satisfies the `RoundTripper` interface. Making `fakeTripper` a function type of the same signature with the `RoundTripper`'s `RoundTrip` method will allow us to convert a function with the same signature of the `RoundTrip` method to a `RoundTripper`. For instance, let's convert the following function to a `RoundTripper`.

```
trip := func(*http.Request) (*http.Response, error) {

    resp := &http.Response{            #A
        StatusCode: http.StatusInternalServerError,    #A
    }                         #A
    return resp, nil
}
fake := fakeTripper(trip)               #B
```

The `trip` function above has the same signature as the `RoundTrip` method. This allows us to convert this function to a `fakeTripper`, satisfying the `RoundTripper` interface. Since `fake` satisfies `RoundTripper`, we can set the HIT client's `Transport` field to this fake tripper.

```
client := &Client{
    Transport: fake,
}
```

The HIT client will use the fake transport to make requests. Because the transport is fake, it lacks the capability to establish connections to a server. Each request attempt will return an internal error, allowing us to test whether the HIT client reports errors in the summary.

Testing with the fake tripper

Now that we have a fake transport, we can test the HIT client with a fake transport as above.

As the next listing shows, we create a fake `RoundTripper` that logs the incoming request (we can remove this once we run the test as it's only for illustration) and returns a response with an internal server error status code. Then, we set the fake transporter to the HIT client.

Listing 8.14 Testing with the fake transporter (client_test.go)

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

    req, err := http.NewRequest(http.MethodGet, "/", http.NoBody)
    if err != nil {
        t.Fatalf("new http request: %v", err)
    }

    fail := func(req *http.Request) (*http.Response, error) {
        t.Logf("fakeTripper: %s", req.URL)                #A

        resp := &http.Response{                      #B
            StatusCode: http.StatusInternalServerError,        #B
        }                            #B

        return resp, nil                        #B
    }

    client := &Client{
        Transport: fakeTripper(fail),                 #C
    }

    sum := client.Do(context.Background(), req, 5)           #D

    if got := sum.Errors; got != 5 {                #D
        t.Errorf("Errors=%d; want 5", got)
    }
}
```

Once we call `Do`, `hit.Client`'s `http.Client` will send requests to the fake transporter, which will respond with an internal server error status code. So, we expect the client to report these errors (since we tell the client to make five requests, we should see five errors).

Trying out the fake tripper

We've written a test that makes five requests with the HIT client to a fake round-tripper that keeps returning internal server errors. In response, the test expects five errors. Once the HIT client's request reaches the fake round-tripper, we should see them in the logs. Let's try.

```
$ go test ./hit -run=TestClientDo -v

=== CONT  TestClientDo
    client_test.go:48: fakeTripper: /    #A
    client_test.go:48: fakeTripper: /    #A
    client_test.go:48: fakeTripper: /    #A
    client_test.go:48: fakeTripper: /    #A
    client_test.go:48: fakeTripper: /    #A
--- PASS: TestClientDo (0.00s)
```

The test has passed. We see the logs from the fake round-tripper for each request the client sends. Remember to remove `Logf` from the fake round-tripper to declutter the test logs.

8.4.3 Wrap up

We wrote two tests, one with a `httptest.Server` and the other with a `RoundTripper`. The latter approach doesn't require us to launch an HTTP server to test the client. Both are valid approaches to testing a client and depend on our preferences. Since the HIT client now allows clients to customize the `Transport` field, users can do more interesting things, such as setting routing requests through a proxy and managing the connections and timeouts. Another use case is to trace requests (see https://go.dev/blog/http-tracing).

- `httptest.Server` is an HTTP server tuned for testing purposes.
- `httptest.NewServer` launches a new test server at a random port.
- `Server.Close` closes the test server.
- `http.HandlerFunc` can convert a function to an `http.Handler`.
- The `atomic` package provides concurrency-safe types.
- `atomic.Int64` is a concurrency-safe 64-bit atomic integer counter.
- `http.RoundTripper` represents a type that can make HTTP requests. The `RoundTrip` method takes a `http.Request` and returns an `http.Response`.
- Passing a `RoundTripper` to `http.Client` can change its request making behavior.

8.5 Exercises

1. Since the throttler slows down the pipeline, it can take a little while for the producer's turn to come to check if the context is canceled. This can cause a delay in the concurrent pipeline stopping. Stop the throttling if the context is canceled.
2. Add the HIT client to the ability to make POST requests.
3. Add an option function to the HIT client for the `Transport` field.
4. Write a struct called `LogTripper` that satisfies `RoundTripper`. This struct should log and forward the incoming request to `http.DefaultTransport`'s `RoundTrip` method. Set `LogTripper` as the HIT client's `Transport` and see if the requests are logged.

8.6 Summary

- The `context` package carries cancellation signals.
- The `http` package allows us to send HTTP requests.
- The `httptest` package has a test server that we can use in tests.
- Option functions pattern allows us to pass arbitrary variadic options to functions.

9 Structuring HTTP Servers

This chapter covers

- Organizing packages for enhanced reusability and clearer code structure.
- Developing, running, and securing HTTP servers for robustness.
- Routing incoming HTTP requests to appropriate handlers efficiently.
- Separating business logic from server logic for cleaner architecture.

This chapter and the next two explain the concepts from the perspective of a fictional company, Bite. A start-up with ambitions to transform the world of link management. We've joined the company as engineers. Our first project is a link server that shortens URLs based on a user-specified short key. When a user requests the shortened URL, the server redirects the user to the long URL (i.e., using the short key `"go"` redirects to `"https://go.dev"`).

Note

Find the source code for this chapter at https://github.com/inancgumus/gobyexample/tree/main/bite.

Getting a preview

The link server will be an HTTP server. Let's explore how it would operate.

We would fire up the server and have it listen for incoming requests like this:

```
$ go run ./cmd/linkd
```

```
INFO starting app=linkd addr=localhost:8080
```

We can now open another command-line session and start sending requests to the server. To shorten a link, we can make a POST request (using our favorite client, like curl).

```
$ curl -i localhost:8080/shorten -d '{"key": "go", "url": "https:
```

```
HTTP/1.1 201 Created
{"key":"go"}
```

The server has created a short link with the key "go.".

Now, we can make another request to resolve the original URL using the key "go."

```
$ curl -i localhost:8080/r/go
```

```
HTTP/1.1 302 Found
Location: https://go.dev
```

If we were requesting with a browser, the browser would automatically redirect this request to "`https://go.dev`". That's all for the initial preview. We're all set to get started now.

Structuring packages for comprehension and usability

Many people new to Go get stuck when organizing packages. Say we have a savvy gopher on our team, Aren. He says Go packages are relatively more comprehensive than other languages. Packages include everything about their domain, keeping types closer.

Idiomatic packages are experts in their specific domains.

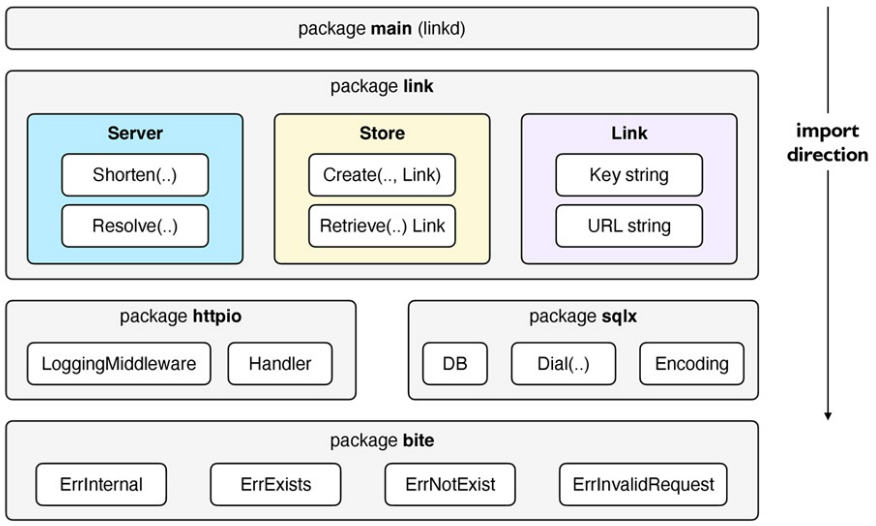
For instance, the `net/http` package has HTTP server, client, router, request, and response types. There are no separate `httpserver`, `httpclient`, and `httprouter` packages. This `http` package imports the `net` package (but not the reverse to avoid circular dependencies). Similarly, `net` offers TCP, UDP, and Unix connection types in the same package.

Note

Circular import is a compile-time error when Package A imports B and B imports A.

Returning to the URL shortener project, Figure 9.1 shows the package structure. This layered package structure (also used by the standard library) avoids circular dependency issues. Like the standard library packages, each package imports packages only from below but not above their level (e.g., `linkd` can import `bite`, but `bite` shouldn't import `linkd`).

Figure 9.1 A layered approach to organizing packages.



We purposely avoid generic and pointless package names like `common`, `util`, `helper`, `api`, `handler`, `controller`, `model`, and `repository`. These often gather unrelated dependencies, making them tricky to maintain. These packages are confused about their domain.

The code should tell us what it does at a glance since we read more code than we write. So, package names should be descriptive and have a clear purpose. For instance, `link` is meaningful and easy to type and remember. Its items have meaningful names, too.

- A link `Server` that can `Shorten` and `Resolve` links.
- A link `Store` to `Create` and `Retrieve` links from the database.
- A `Link` with a `Key` and a `URL` to redirect to.

Notice how easy it is to make natural sentences with these types. They're also in the same package and near where they're used. All this makes `link` tell a meaningful story and makes it easy to guess and find related code. The package name also helps with readability.

For instance, `link.Store` is readable, clear, and easy to type when used from other packages. However, extensive naming (e.g., `link.ShortenedLinkStore`) doesn't automatically improve understanding. The documentation can provide these extra details.

```
package link

// Store persists links.
type Store struct { .. }
```

Organizing packages in a layered structure avoids circular dependency issues, and proper naming enhances clarity, understanding, readability, and usage. Still, we should remember that there's no single right way. We aren't restricted to a particular package structure as long as we avoid import cycles. Starting with a package and gradually expanding it to others is fine. By embracing the YAGNI (You Aren't Gonna Need It) and KISS (Keep it Simple and Stupid) principles, we can create effective packages that are easy to use and understand.

Tip

Think of packages as providers of functionality rather than containers of functionality. Packages are not storehouses where we can put everything. Think of them as boutique shops that provide niche services to their clients. The package's API is its storefront.

9.1 Server, Handler, and HandlerFunc

Since the link server will be an HTTP server, we'll dive deep into how to craft an HTTP server using Go's `net/http` package. We'll first learn about the basic concepts of the `net/http` package. This package helps us run an HTTP server to listen to client connections and serve requests. Lastly, we'll take some precautions to make it more secure.

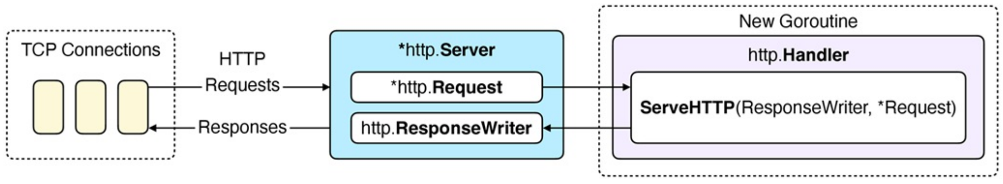
9.1.1 Basics

The `net/http` package has the following types that we can use to run HTTP servers:

- `http.Server`: Listens for client connections.
- `http.Handler`: Serves HTTP requests.
- `http.ResponseWriter`: Responds to the client.
- `http.Request`: Contains request details, such as the requested URL.

Figure 9.2 illustrates serving HTTP requests using `http.Server` and an `http.Handler`. Once we launch `http.Server`, it spawns a goroutine for each incoming request. This goroutine calls the registered handler's `ServeHTTP` method, passing a `*Request` and `ResponseWriter`.

Figure 9.2 Server routes client requests to a Handler.



Inside an handler, we can use `ResponseWriter` to respond to the client:

```
type ResponseWriter interface {
    Header() Header        #A
    Write([]byte) (int, error)    #B
    WriteHeader(statusCode int)    #C
}
```

And, use `*Request` to inspect request details, such as requested URL path, etc. For instance:

```
notFound := func(w http.ResponseWriter, r *http.Request) {

    // use r to read request details. for instance:
    _ = r.URL.Path     // returns the requested URL path

    // use w to respond to the client. for instance:
    w.WriteHeader(404) // sends an HTTP status code 404
}
```

Now that we looked at how to serve requests and respond to clients, let's look at how to start a server. We can use the `ListenAndServe` function to start a default `Server`. It takes a network address to listen for TCP connections and a `Handler` to serve HTTP requests.

```
func ListenAndServe(addr string, handler http.Handler) error
```

However, our `notFound` function isn't a `Handler` yet because it doesn't have a `ServeHTTP` method. We can't pass it to `ListenAndServe`. Let's learn to make this function a `Handler`.

The Handler interface and the HandlerFunc type

`Handler` is an elegant interface with a single `ServeHTTP` method:

```
type Handler interface {

    ServeHTTP(ResponseWriter, *Request)
}
```

We could declare a type that satisfies `Handler` by implementing `ServeHTTP` like this:

```
type myHandler struct{}

func (myHandler) ServeHTTP(w ResponseWriter, r *Request) {
    notFound(w, r)                        #A
}
```

However, creating a type isn't always practical. Instead, we can convert our function to a `Handler` using `HandlerFunc`. It has a `ServeHTTP` method and satisfies `Handler`.

```
type HandlerFunc func(ResponseWriter, *Request)            #A

func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
    f(w, r)                               #C
}
```

Let's convert `notFound` to `HandlerFunc`. Then, we can pass the result to `ListenAndServe`:

```
notFound := func(w http.ResponseWriter, r *http.Request) { /* ..
handler  := http.HandlerFunc(notFound)                #A
ListenAndServe("localhost:8080", handler)               #B
```

In summary, we can either convert a regular function that takes a

`ResponseWriter` and a `*Request` to a `HandlerFunc` or create a type with a `ServeHTTP` method to make it satisfy `Handler`. Then, we can register the handler on a `Server` to serve HTTP requests (or pass a `Handler` to `ListenAndServe` to serve with a default `Server`).

First steps toward the server

Having looked at the `net/http` package basics, let's run another server. Similar to what we've done, we'll again use `ListenAndServe`. This will be the basic skeleton of the link server. It won't shorten and resolve links yet but will be a good start for us.

As the next listing shows, we're first setting a logger for the program using the `slog` package. Right after, we're converting a function to a `Handler` using `HandlerFunc`. Lastly, we're launching an HTTP server to listen for client connections on `localhost:8080`.

Listing 9.1 Running the server (cmd/linkd/linkd.go)

```
package main

import (
    "errors"
    "fmt"
    "log/slog"
    "net/http"
    "os"
)

func main() {
    const addr = "localhost:8080"

    log := slog.With("app", "linkd")                #A
    slog.SetDefault(log)                        #A

    log.Info("starting", "addr", addr)

    links := func(w http.ResponseWriter, r *http.Request) {    #B
        fmt.Fprintln(w, "hello from the bite links server!")
    }
    handler := http.HandlerFunc(links)                #D
    err := http.ListenAndServe(addr, handler)            #E
    if !errors.Is(err, http.ErrServerClosed) {            #F
        log.Error("server closed unexpectedly", "message", err)
    }
}
```

Note

The `ListenServe` function blocks until the server stops and always returns an error. The `ErrServerClosed` error is expected and we don't need to log an error when it happens.

The server routes requests to `handler`, which internally calls `links`. Our function receives a `ResponseWriter` (an `io.Writer`), and it writes a hello message to clients.

Let's run this program to launch the server and serve requests.

```
$ go run ./cmd/linkd
```

```
INFO starting app=linkd addr=localhost:8080    #A
```

Since the server waits for incoming connections, we can send a request to the server from another terminal using any favorite HTTP client (it's also possible to use a browser):

```
$ curl localhost:8080
```

```
hello from the bite links server!
```

We've launched a server, which handles the request using the `links` function. Next, we'll look at launching a customized server, such as a more robust one with a timeout.

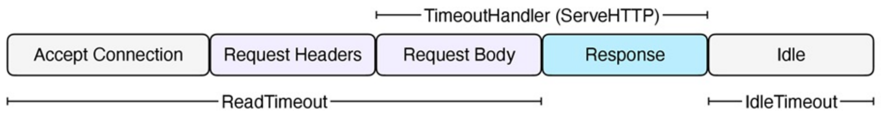
Warning

Remember to stop the server with CTRL+C (or Command+C on a Mac) and rerun it whenever you make changes to the code. Otherwise, the server will run the old code.

9.1.2 Hardening the server

Imagine the server crashing due to many client requests that took too long to read or handlers taking too long to return. To avoid these problems, we can set up timeouts. Figure 9.3 shows the timeout settings we can configure and which time intervals they are for.

Figure 9.3 Request and response lifecycle and timeout intervals.



Here, we have two server timeout settings: `ReadTimeout` and `IdleTimeout`:

- `ReadTimeout` starts ticking from accepting a connection until the request is read.
- `IdleTimeout` keeps the connection alive and reuses it if a new request comes in.

Since `ListenAndServe` doesn't allow setting timeouts, we must create a custom `Server`:

```
srv := &http.Server{
    ReadTimeout: 20 * time.Second,
    IdleTimeout: 40 * time.Second
    // ...
}
```

In the figure, we also see `ServeHTTP` (which is not a server timeout setting). Its duration starts from the server's calling its handler's `ServeHTTP` method until the method returns. One option to protect the server from slow handlers is to use `TimeoutHandler`:

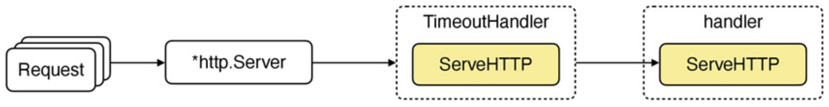
```
// TimeoutHandler wraps h and returns a timeout handler.
func TimeoutHandler(h http.Handler, dt time.Duration, msg string)
```

It takes a handler and returns another that can timeout. For instance, we can wrap our current handler in a timeout handler and register on the server:

```
srv := &http.Server{
    ...
    Handler: http.TimeoutHandler(handler, time.Second, "timeout")
}
```

Figure 9.4 shows how the request routing works when we wrap the handler this way.

Figure 9.4 TimeoutHandler reroutes the request to the handler.



Once a request is received, the timeout handler will get it before the wrapped handler. If the wrapped one takes more than one second, the timeout handler will timeout, cancel the request's `Context`, and write a "timeout" message and a 503 (Service Unavailable).

Note

Handlers should pass the request's context to long-running operations to tell them to cancel their work when the time comes. Otherwise, these operations will keep running.

Now that we have looked at our timing options, let's update the server in the next listing.

Listing 9.2 Setting timeouts (cmd/linkd/linkd.go)

```
func main() {
    const timeout = 10 * time.Second
    ...

    handler := http.HandlerFunc(links)
    handler  = http.TimeoutHandler(handler, timeout, "timeout")

    srv := &http.Server{
        Addr:          addr,                        #B
        Handler:       handler,                        #C
        ReadTimeout:   timeout * 2,                        #D
        IdleTimeout:   timeout * 4,
    }
    if err := srv.ListenAndServe(); !errors.Is(err, http.ErrServe
        log.Error("server closed unexpectedly", "message", err)
    }
}
```

Our server is more resilient than before. While the server timeout settings will protect the server from slow or malicious clients, the timeout handler will protect the server from slow handlers. Although we don't have slow handlers yet, we're readying our server for the future.

Note

We should configure timeouts depending on our needs. Here, we set them arbitrarily.

However, despite its advantages, `TimeoutHandler` disables some functionalities of the underlying `ResponseWriter`, such as sending streaming data or the ability to take over the underlying TCP connection (which is useful for upgrading the connection to a WebSocket).

It only keeps the server push functionality (which is useful for HTTP/2 servers). Lastly, it buffers what the wrapped handler writes into an in-memory buffer. So, in a real-world scenario, it might make more sense to use this middleware on a per-handler basis (rather than setting it as a global server's handler). Luckily, as of Go 1.20, we can now set individual timeouts inside handlers. We'll look into this later in the chapter.

Using HTTPS

To protect against man-in-the-middle attacks, we can launch the server using the `ListenAndServeTLS` method (similar to `ListenAndServe`) to listen and respond over HTTPS connections. Visit https://pkg.go.dev/net/http#Server.ListenAndServeTLS to learn more.

Still, running a server over HTTP and using a reverse proxy to handle HTTPS is common. A reverse proxy can manage TLS termination, decrypt HTTPS requests and pass them as HTTP to the server that's listening for HTTP. This can simplify both development and HTTPS setup.

9.1.3 Wrap up

- `Server` listens and serves incoming HTTP requests with a `Handler`.
- `Handler` is an interface that has a `ServeHTTP` method.
- `HandlerFunc` can convert a function to a `Handler`.
- `ResponseWriter` can write the handler's response to the client.
- `Request` contains request-related information.
- `ListenAndServe` launches a server and blocks until the server stops.
- `TimeoutHandler` wraps a handler and returns another that can timeout. When a timeout occurs, the request's `Context` is canceled, and the client sees an error.

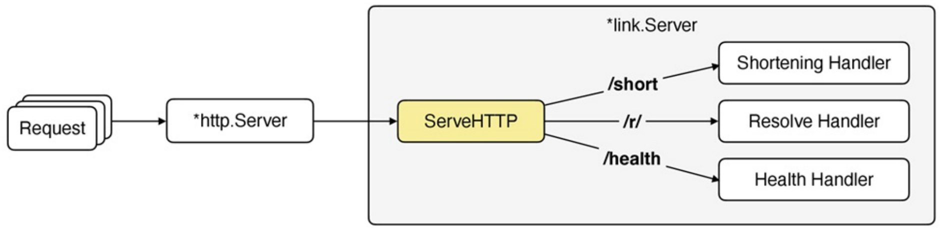
Note

For more information about the `net/http` package, visit: https://pkg.go.dev/net/http.

9.2 Request routing

An HTTP server typically has multiple endpoints and routes incoming requests to them. However, `Server` *only allows for registering a single handler*. Still, as Figure 9.5 shows, we can craft a `Handler` type that can route requests to other handlers based on a route.

Figure 9.5 The handler routes requests to the inner handlers.



Note

We don't show the timeout handler in the figure to keep the explanations to the point.

This way, we can register this handler (i.e., `link.Server`) on the HTTP server (i.e., `http.Server`) that can route requests to the handlers based on their routes. For instance, in Figure 9.5, when a request comes to "/short", the HTTP server routes the request to our handler, which in turn reroutes the request to the shortening handler.

We may observe that only the resolve route (/r/) has a trailing slash (/). URL patterns ending with / act as prefix patterns, matching any URL path starting with that pattern and additional segments. This helps manage multiple URLs sharing a common prefix. We use the trailing slash to match any path starting with /r/ followed by a key (/r/shortkey1, /r/shortkey2, etc.). Conversely, the other routes lack /, as they only match the precise URL path.

9.2.1 Crafting a custom router

The next listing shows the implementation of the `link.Server` type. It implements the `Handler` interface and has a `ServeHTTP` method (which will be called for all incoming requests). `ServeHTTP` routes incoming requests to the handlers based on their routes.

Listing 9.3 Implementing the server (link/server.go)

```
package link


import (
    "net/http"
    "strings"
)

// Server is a link management server.
type Server struct {
    // fields can be added to store server-specific dependencies.
}

// NewServer creates and returns a new Server.
func NewServer() *Server {
    return &Server{}
}

func (s *Server) ServeHTTP(w http.ResponseWriter, r *http.Request
    switch p := r.URL.Path; {                              #B
    case p == "/shorten":                                  #B
        Shorten(w, r)                                  #B
    case strings.HasPrefix(p, "/r/"):                          #B
        Resolve(w, r)                                  #B
    case p == "/health":                                 #B
        Health(w, r)                                  #B
    default:
        http.NotFound(w, r) // responds with 404 if no path match
    }
}
```

`Server` doesn't have any fields yet (it's an empty struct), but we'll add them soon. Once `ServeHTTP` finds a matching route path for a handler, it routes the incoming request to that handler (e.g., "/r/" goes to `Resolve`). It passes the current `ResponseWriter` and `Request` to the handler so that the handler can take over the processing of the request. We also call `NotFound` if a route isn't found (responds with a status code of 404 (Not Found)).

9.2.2 Crafting the handlers

Now that we've implemented the link server, next are the handlers. See the next listing.

Listing 9.4 Implementing the handlers (link/server.go)

```
package link


import (
    "fmt"
    "net/http"
)

func Shorten(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusCreated)                      #A
    fmt.Fprintln(w, "go")                              #B
}

func Resolve(w http.ResponseWriter, r *http.Request) {
    const uri = "https://go.dev"
    http.Redirect(w, r, uri, http.StatusFound)            #C
}
```

```
func Health(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "OK")                         #B
}
```

We're building the skeleton of our handlers. They don't do much currently other than responding with status codes and some text, like "go". The resolve handler also redirects the request. Ours may not be the most advanced routing mechanism and the handlers have hard-coded responses. Still, this basic structure is a good starting point for building out the full functionality later. Next, we'll integrate our `link.Server` with `http.Server`.

Note

The `link` package makes it easy to read and understand what it offers (e.g., `link.Shorten` and `link.Resolve`). Readable code is critical for a package's usability.

By default, handlers write a status code of OK

The health check handler, in particular, won't write a status header using `WriteHeader`, but it will still respond with an HTTP status code of `OK`. Handlers will automatically do that if we don't specify a different status code using the `WriteHeader` method before calling `Write`.

9.2.3 Routing with the custom router

We've created a server, incorporated a router, and linked the handlers. It's time to register `link.Server` on an `http.Server`. As the next listing shows, we're creating a new server using `NewServer` and passing the result as a `Handler` to `TimeoutHandler`.

Listing 9.5 Integrating with the HTTP server (cmd/linkd/linkd.go)

```
func main() {
    ...

    links := link.NewServer()                     #A
    handler := http.TimeoutHandler(links, timeout, "timeout")
    srv := &http.Server{
        Addr:        addr,
        Handler:     handler,
        ReadTimeout: timeout,
    }
    if err := srv.ListenAndServe(); !errors.Is(err, http.ErrServe
        log.Error("server closed unexpectedly", "message", err)
    }
}
```

Since `link.Server` has a `ServeHTTP` method, we can use this method as a `Handler`. The `ServeHTTP` method will receive and route incoming requests to handlers. Let's give it a try:

```
$ curl -i localhost:8080/shorten

HTTP/1.1 201 Created            #A
go                   #A

$ curl -i localhost:8080/r/go
HTTP/1.1 302 Found            #B
Location: https://go.dev            #B

$ curl -i localhost:8080/health
HTTP/1.1 200 OK                #C
OK                   #C

$ curl -i localhost:8080/nope
HTTP/1.1 404 Not Found            #D
404 page not found            #D
```

With the implementation of the link server and the router, incoming requests now effectively route to the appropriate handlers based on requested URL paths.

9.2.4 Wrap up

- `Server` only allows a single `Handler` to be registered.
- A handler can route requests to multiple handlers.
- `WriteHeader` writes an HTTP header to the client (e.g., a status code).
- `Redirect` returns an HTTP status code of 302 (Found) and a Location header.

9.3 Multiplexing with ServeMux

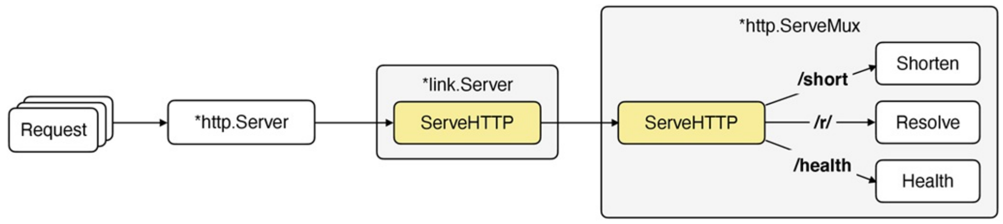
The custom router we implemented can become a hassle when we want to add many routes. To avoid scaling issues and manually route requests to handlers, we can use `ServeMux` (also called a *multiplexer*, *mux*, or *muxer*) to register the handlers and handle the routing.

Note

`ServeMux` has a `ServeHTTP` method and thus is a `Handler`.

As Figure 9.6 shows, `ServeMux` will work similarly to our custom router. When a request arrives, `link.Server`'s `ServeHTTP` is invoked. This, in turn, calls `ServeMux.ServeHTTP` to forward the request to a handler corresponding to the registered route pattern.

Figure 9.6 Server calls ServeMux's ServeHTTP to route requests to handlers.



With `ServeMux`, we no longer need a custom muxer. See the next listing.

Listing 9.6 Routing with ServeMux (link/server.go)

```
type Server struct {
    mux *http.ServeMux                #A
}

func NewServer() *Server {
    mux := http.NewServeMux()
    mux.HandleFunc("POST /shorten", Shorten)    #B
    mux.HandleFunc("GET /r/", Resolve)        #C
    mux.HandleFunc("GET /health", Health)        #C

    return &Server{
        mux: mux,
    }
}

func (s *Server) ServeHTTP(w http.ResponseWriter, r *http.Request
    s.mux.ServeHTTP(w, r)                       #D
}                                   #D
```

We're creating a new `ServeMux` and then registering our handlers. The `HandleFunc` method converts each function to a `Handler` (e.g., `Shorten` becomes a `Handler`). Here, we use a set of novel routing patterns (included as of Go version 1.22):

- "POST /shorten" only allows POST requests to the "/shorten" route.
- "GET /r/" only allows GET requests to the "/r/" route.

By converting each handler function into a `Handler` using `HandleFunc` and calling the `ServeHTTP` method of `ServeMux`, we're delegating routing responsibilities to `ServeMux`. Now, `ServeMux` can route incoming requests to the `link.Server`'s handlers. This frees us from handling the routing work. We'll streamline this code even more later. Let's give it a try.

```
$ curl -i localhost:8080/shorten -d "something"

HTTP/1.1 201 Created
go
$ curl -i localhost:8080/r/go
HTTP/1.1 302 Found
Location: https://go.dev
$ curl -i localhost:8080/health
HTTP/1.1 200 OK
OK
$ curl -i localhost:8080/nope
HTTP/1.1 404 Not Found
404 page not found
$ curl -i localhost:8080/shorten -XGET    #A
HTTP/1.1 405 Method Not Allowed
method not allowed
```

Note

For more information about `ServeMux`, visit https://pkg.go.dev/net/http#ServeMux. Remember that we use `HandleFunc` to register a handler rather than `HandlerFunc`. The prior is a method on `ServeMux`, while the latter is a type to convert a function to a `Handler`.

9.3.1 Embedded fields

When a struct type embeds another type *without a field name*, the embedded type's methods and fields become *promoted* and directly usable on the struct type. Embedding can avoid duplicate code and keep things neat and tidy. Let's look at a usage example.

In the previous code, `ServeHTTP` did nothing but called `ServeMux`'s `ServeHTTP`.

```
package link

type Server struct {
    mux *http.ServeMux
}

func (s *Server) ServeHTTP(w http.ResponseWriter, r *http.Request
    s.mux.ServeHTTP(w, r)
}
```

Instead, we can remove this method, and *embed* `Handler` (without a field name) as follows:

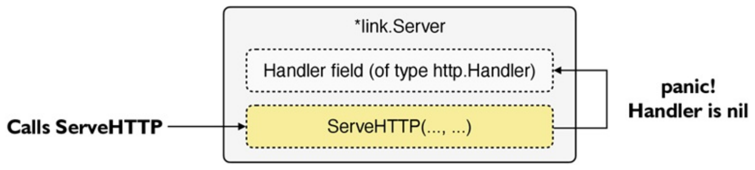
```
package link

type Server struct {
    http.Handler        #A
}
```

Note

To promote a type's method and fields, we must embed it without a field name.

Once we embed `Handler`, as Figure 9.7 shows, `Server` gets a promoted `ServeHTTP` method and a `Handler` field (embedded type's name), hence still satisfying the `Handler` interface.

Figure 9.7 Server gets a Handler field and a promoted ServeHTTP method. Since the field is nil, calling ServeHTTP on Server calls Handler.ServeHTTP and panics.



Embedding an interface doesn't come with an implementation. Calling the promoted methods will panic at runtime unless we set an implementation. Calling `ServeHTTP` will panic:

```
var s Server
s.ServeHTTP(.., ..) // panics: nil pointer dereference
```

We can avoid this by setting the `Handler` field to a `Handler` implementation like `ServeMux`. Let's look at the next listing for the complete implementation.

Listing 9.7 Using interface embedding (link/server.go)

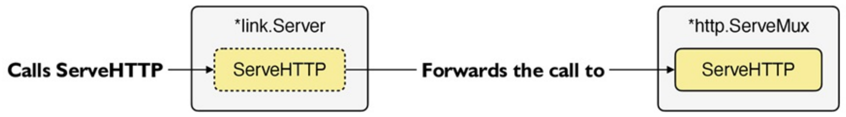
```
type Server struct {
    http.Handler                    #A
}

func NewServer() *Server {
    mux := http.NewServeMux()
    mux.HandleFunc("POST /shorten", Shorten)
    mux.HandleFunc("GET /r/", Resolve)
    mux.HandleFunc("GET /health", Health)

    return &Server{
        Handler: mux,                #B
    }
}
```

We remove the `ServeMux` method and set the `Handler` field to `mux` (`ServeMux`) to avoid panics. This allows `ServeMux` to operate as before and route requests to the handlers.

We can assign `mux` to the `Handler` field because `mux` (of type `*ServeMux`) satisfies the `Handler` interface. We could assign another `Handler` implementation to this field as well. As Figure 9.8 shows, now calling

`Server`'s promoted `ServeHTTP` will call `ServeMux`'s.

Figure 9.8 Calling Server.ServeHTTP calls Server.Handler.ServeHTTP.



Embedding promotes (adds) the embedded type's methods and fields on the embedding type. Embedding an interface allows the outer type to satisfy all the interfaces the embedded type satisfies. However, embedding an interface doesn't come with an implementation.

It causes panic when the promoted methods are called unless a concrete implementation of that interface is assigned to the embedded field. Embedding is a nice syntactic sugar to reduce duplication and helps the composition of types. For more information about embedding, visit go.dev/doc/effective_go#embedding. Also, see the sidebar for its pitfalls.

Dive deep: Be careful when embedding a type

Embedding a type can unintentionally export the embedded type's methods from the embedding type, leading to potential issues like encapsulation breaches. Be careful when embedding types to prevent unintended method exports. In our case, `link.Server` got an exported `Handler` field since the `Handler` type is exported. Since we want `link.Server` to be used as a `Handler` in `http.Server`, having an exported `Handler` field is what we want.

Also, embedding an interface can hide the methods of the underlying type. For instance, in our case, `*ServeMux` is the `Handler` field's underlying type. We can extract `*ServeMux` from the field using type assertion (as the field is an interface, not `*ServeMux`). For example:

```
s := NewServer()
mux := s.Handler.(*http.ServeMux) // mux is the *ServeMux NewServ
```

We can extract `*ServeMux` from the `Handler` field because its underlying type is `*ServeMux` (because that's what we assign to the field (see Listing 9.7)). However, if we pass `s` to a function that expects a `Handler`, that function can never extract `ServeMux`:

```
func extract(h http.Handler) {
    _ = h.(*http.ServeMux)
}
extract(s)          // panic: .. s is not *http.ServeMux
extract(s.Handler)  // ok: s.Handler is *ServeMux
```

This is because this Handler's underlying type is *link.Server, not *http.ServeMux. `*Server` contains `*ServeMux` in its `Handler` field, but `*Server` itself is not a `*ServeMux`. This is typically what we would expect. But we'll soon see when this becomes a problem.

9.3.2 Retrieving named wildcard segments

Go 1.22 has added support for matching routes on wildcard patterns. This, for instance, allows us to retrieve the short link key from the URL path:

```
$ curl -i localhost:8080/r/go

...
```

Here, the short link key is "go," and it's in the URL path. We can retrieve the key using `Request`'s `PathValue` method once we add the necessary route to `ServeMux`. The next listing modifies the resolve handler's route to "/r/{key}." Think of "{key}" as a template. For example, using `PathValue`, we can extract "go" from "/r/go" or "rusty" from "/r/rusty."

Listing 9.8 Including a wildcard segment (link/server.go)

```
func NewServer() *Server {
    mux := http.NewServeMux()
    mux.HandleFunc("POST /shorten", Shorten)
    mux.HandleFunc("GET /r/{key}", Resolve)    #A
    mux.HandleFunc("GET /health", Health)

    return &Server{
        Handler: mux,
    }
}
```

Now that we've set up the resolve route, we can retrieve the key using

`PathValue`. We're passing the wildcard name. The next listing updates the resolve handler.

Listing 9.9 Extracting the key with PathValue (link/server.go)

```
func Resolve(w http.ResponseWriter, r *http.Request) {    #A
    fmt.Fprintf(w, "key: %s\n", r.PathValue("key"))        #B

    const uri = "https://go.dev"
    http.Redirect(w, r, uri, http.StatusFound)
}
```

Everything is in place. Let's give it a try.

```
$ curl -i localhost:8080/r/go

key: go

$ curl -i localhost:8080/r/rusty
key: rusty
```

Although it works, writing to a client before the redirect will cause the server to print a warning: "http: superfluous response.WriteHeader call." This is because writing to a client (`Fprintf` calls `ResponseWriter`'s `Write` method and hence writes to the client) sends an HTTP status code of 200 (OK) in an HTTP header. Then, the redirect handler also writes another header to redirect the client. We'll remove this superfluous `Fprintf` soon.

Note

For more information on routing, visit: go.dev/blog/routing-enhancements

9.3.3 Wrap up

- `ServeMux` is a `Handler` that can route requests to other handlers.
- Route patterns can be set only to allow specific HTTP methods like GET.
- Wildcard segments can be defined with the format "{name}".
- `Request`'s `PathValue` retrieves a wildcard segment from the URL path.
- Embedding promotes the fields and methods of a type to the embedding

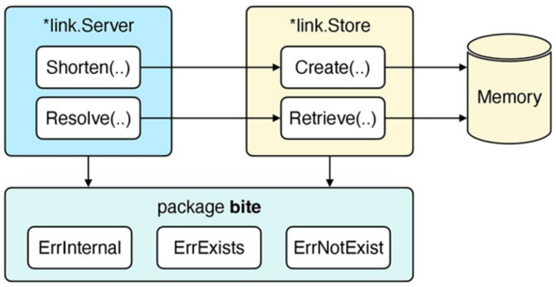
type.

9.4 Business logic

Now that we've implemented the link server, let's implement the business logic.

As Figure 9.9 shows, we define common errors in the `bite` package. The business rules and storage related to link management go into the `link` package. `Store` stores links in a temporary in-memory map (we'll connect it to a database in the following chapters).

Figure 9.9 Handlers use Store to persist and retrieve links. The bite package defines the common errors that we use throughout the project.



We'll start with defining common errors, then business rules, and lastly, the storage type. We improve our codebase's maintainability, clarity, and reuse by isolating business rules and cross-cutting concerns. All packages in our company can use the same error definitions.

The next listing shows the declaration of these common errors.

Listing 9.10 Defining common errors (errors.go)

```
package bite


import "errors"

var (
    ErrExists         = errors.New("already exists")
    ErrNotExist       = errors.New("does not exist")
    ErrInvalidRequest = errors.New("invalid request")
    ErrInternal       = errors.New(
        "internal error: please try again later or contact suppor
    )
)

// other shared types—structs, interfaces, etc., may be in differ
```

Like the rest of our packages, the server handlers in the `link` package will return and process these errors, even augment them with additional information when needed. Besides error definitions, anything cross-cutting can be included in `bite` (e.g., user management).

9.4.1 Core business logic

Our next goal is to implement the core business logic of our link management project.

We'll have a `Link` type to represent links. We also have validation functions that check the business rules while creating and retrieving links. The storage logic we'll add shortly will use these functions. Check out the next listing for the implementation.

Listing 9.11 Adding business rules (link/link.go)

```
package link


import (
    "errors"
    "fmt"
    "net/url"
    "strings"
)

// Link represents a link.
type Link struct {
    Key string
    URL string
}

// validateNewLink checks a new link's validity.
func validateNewLink(link Link) error {
    if err := validateLinkKey(link.Key); err != nil {
        return err
    }
    u, err := url.Parse(link.URL)
    if err != nil {
        return err
    }
    if u.Host == "" {
        return errors.New("empty host")
    }
    if u.Scheme != "http" && u.Scheme != "https" {
        return errors.New("scheme must be http or https")
    }
    return nil
}

// MaxKeyLen is the maximum length of a key.
const MaxKeyLen = 16

// validateLinkKey checks the key's validity.
func validateLinkKey(key string) error {
    if strings.TrimSpace(key) == "" {
        return errors.New("empty key")
    }
    if len(key) > MaxKeyLen {
        return fmt.Errorf("key too long (max %d)", MaxKeyLen)
    }
    return nil
}
```

The validation functions ensure only valid links can be created and retrieved. We can now integrate `Link` and the validation functions into the storage service.

9.4.2 Storage service

Our next goal is to provide a storage service for the server handlers to store and retrieve shortened links. The `Store` type stores links in a volatile in-memory map (i.e., once the server shuts down or restarts, the links will be gone).

We'll later upgrade it to work with an SQL database to persist links.

Data storage and retrieval

The next listing shows the implementation of the `Store` type.

Listing 9.12 Adding a storage service (link/Store.go)

```
package link


import (
    "context"
    "fmt"
    "sync"

    "github.com/inancgumus/gobyexample/bite"
)

// Store persists and retrieves links.
type Store struct {
    mu    sync.RWMutex            #A
    links map[string]Link            #B
}

// NewStore returns a new Store.
func NewStore() *Store {
    return &Store{
        links: make(map[string]Link),    #C
    }
}
```

In `Store`, we have a map of strings and links. Inside the `NewStore` constructor, we initialize this map. Otherwise, writing to the map would panic. We also have a `sync.RWMutex` field. We also have the `NewStore` constructor to initialize the map field.

The server handlers run concurrently inside goroutines. So, we use a `RWMutex` (Read/Write mutex) to protect `links` from uncontrolled concurrent access. `RWMutex` is similar to a `Mutex`, but rather than blocking all, it allows multiple readers to read unless there is a writer. The readers and other writers must wait if there is a writer. We'll use these `RWMutex` methods:

- `Lock` to take exclusive writing access to the mutex.

- `RLock` to take reading access to the mutex.
- `Unlock` to unlock the mutex.

We picked `RWMutex` instead of `Mutex` because our map will be read-heavy rather than write-heavy. We picked `RWMutex` based on intuition, but we should benchmark it in real life.

Note

Mutexes are a better choice than channels if we're after protecting a simple state but not after orchestrating goroutines. `Store` doesn't manage goroutines (instead, goroutines use it). Depending on the situation, mutexes are typically more performant than channels as channels build upon mutexes and need more infrastructural needs than mutexes. To learn more about mutexes, visit https://pkg.go.dev/sync#Mutex and https://pkg.go.dev/sync#RWMutex.

Create and Retrieve

The next listing shows the implementation of the `Create` and `Retrieve` methods.

Listing 9.13 Adding Create and Retrieve methods (link/Store.go)

```
// Create persists the given link.
func (s *Store) Create(_ context.Context, link Link) error {
    if err := validateNewLink(link); err != nil {
        return fmt.Errorf("%w: %w", bite.ErrInvalidRequest, err)
    }
    if link.Key == "fortesting" {                       #C
        return fmt.Errorf("%w: db at IP ... failed", bite.ErrInte
    }

    // holds the write-lock until the function returns
    s.mu.Lock()                                     #D
    defer s.mu.Unlock()                                 #E

    if _, ok := s.links[link.Key]; ok {                     #F
        return bite.ErrExists
    }
    s.links[link.Key] = link                            #G
    return nil
}

// Retrieve gets a link from the given key.
func (s *Store) Retrieve(_ context.Context, key string) (Link, er
    if err := validateLinkKey(key); err != nil {
        return Link{}, fmt.Errorf("%w: %w", bite.ErrInvalidReques
    }
    if key == "fortesting" {
        return Link{}, fmt.Errorf("%w: db at IP ... failed", bite
    }

    // holds the read-lock until the function returns
    s.mu.RLock()                                    #H
    defer s.mu.RUnlock()                                #H

    link, ok := s.links[key]
    if !ok {
        return Link{}, bite.ErrNotExist
    }

    return link, nil
}
```

We're adding `Create` to create a link and `Retrieve` to retrieve a link by its short key. Both take a `Context`, but it won't be useful until the following chapters' database integration (so, we skip its name with a blank identifier). Let's explain how they work.

`Create` validates the link and returns a wrapped error if it's invalid. This way, handlers can check a specific error in the chain of errors. For example, if a link's URL is missing a host part, `Create` would return "invalid request: empty host." The "invalid request" part comes from the wrapped `ErrInvalidRequest`, which handlers can then check using `errors.Is`.

`Retrieve` is similar to `Create` but validates the key and returns a link. Like `Create`, `Retrieve` returns an error for short keys "`fortesting`" to do manual testing.

We're protecting `links` with a read/write mutex. `Create` takes a write lock because it modifies `links`, while `Retrieve` only takes a read lock. Multiple `Retrieve` calls can run simultaneously by many goroutines unless `Create` acquires the mutex.

Tip

Using "defer" to unlock mutexes reduces the chance of forgetting to unlock.

9.4.3 Wrap up

While the `bite` package provides common errors for the entire project, the `link` package deals with link management. The business logic and storage are now ready.

- Shared errors allow the entire project to speak the same language.
- Writing to a map panics if it's not been initialized (either with `make` or a map literal). Reading from a nil map won't panic and returns the element type's zero value.
- `RWMutex` is similar to a `Mutex` but allows multiple readers unless a writer.

Exercise

Improve `Create` to remove very old links from the map. Otherwise, the map could grow so much that it can crash the machine due to out-of-memory errors. Feel free to resurrect your old computer science books to implement advanced invalidation logic.

9.5 Higher-order handler functions

We want the shortening and resolve handlers to use `Store` to create and retrieve links. However, a typical handler signature doesn't allow defining additional input parameters:

```
func Shorten(w http.ResponseWriter, r *http.Request)    #A
func Resolve(w http.ResponseWriter, r *http.Request)    #A
```

We can't register them on an `http.Server` as a `Handler` if we break their signatures. One solution to this issue is to use a struct, and the other one is to use higher-order functions.

The first solution is to attach handlers to a struct and adding a `Store` field:

```
type handler struct           { links *Store     }
func (h *handler) Shorten(..) { ..use h.links here.. }
```

Another solution is to make the handlers higher-order functions. For instance, `Resolve` below takes a `*Store` and returns a `HandlerFunc` closure that we can register on a `ServeMux`.

```
func Resolve(links *Store) http.HandlerFunc {            #A

    return func(w http.ResponseWriter, r *http.Request) {
        /* handler code that uses the links input */
    }
}
```

Then, we can call `Resolve` like this:

```
handler := Resolve(NewStore())        #A
// register handler on the server...
```

Both solutions are fine, but keeping the handlers independent lets us decouple them from `link.Server` and test them directly without having to create a `Server` in the tests every time we want to test a handler. Also, using higher-order functions makes what handlers need to work transparent: `Store`. So, we'll pick the second route.

9.5.1 Resolve handler

The first handler we'll update is the resolve handler. Check out the next listing.

Listing 9.14 Updating the resolve handler (link/server.go)

```
func Resolve(links *Store) http.HandlerFunc {                #A

    return func(w http.ResponseWriter, r *http.Request) {
        link, err := links.Retrieve(r.Context(), r.PathValue("key
        if err != nil {
            httpError(w, err)                           #C
            return                               #D
        }

        http.Redirect(w, r, link.URL, http.StatusFound)
    }
}
```

`Resolve` takes a `Store` and returns a `HandlerFunc`. The returned function extracts the short link key from the request path, queries the database for the key, and redirects the client to a long URL. It also handles errors using the `httpError` function that we'll develop later.

Here, we pass the request's `Context` to `Retrieve`. This way, `Retrieve` can cancel its operation if the client cancels the request (or for another cancellation reason). For instance, `TimeoutHandler` we used earlier cancels this context when a timeout occurs.

Warning

Even if `TimeoutHandler` can return a timeout error to the client, a handler will still continue processing the request if we don't pass (or check) `Context` downstream.

9.5.2 Shortening handler

The last handler we'll update is the shortening handler. See the next listing.

Listing 9.15 Updating the shortening handler (link/server.go)

```
func Shorten(links *Store) http.HandlerFunc {

    return func(w http.ResponseWriter, r *http.Request) {
        link := Link{
            Key: r.FormValue("key"),            #A
            URL: r.FormValue("url"),            #A
        }
        if err := links.Create(r.Context(), link); err != nil {
            httpError(w, err)
            return
        }

        w.WriteHeader(http.StatusCreated)
        w.Write([]byte(link.Key))                #B
    }
}
```

The handler gets a POST request, extracts a short link key and a long URL from the request, creates the link using `Store`, and returns an HTTP status code of 201 (Created). We use `FormValue` because the POST data comes from the request's body. It's encoded as "application/x-www-form-urlencoded" and the `http` package can parse this data.

```
link := Link{
    Key: r.FormValue("key"),
    URL: r.FormValue("url"),
}
```

The last lines write an HTTP status code of 201 (Created) and the key to the client.

Note

Remember to return from a handler after an error to stop processing the request.

9.5.3 Converting errors to HTTP

Handlers use `httpError` to convert errors to HTTP status codes. See the next listing.

Listing 9.16 Converting errors to HTTP (link/server.go)

```
func httpError(w http.ResponseWriter, err error) {
    if err == nil { // no error                #A
        return
    }
    var code int
    switch {
    case errors.Is(err, bite.ErrInvalidRequest):
        code = http.StatusBadRequest
    case errors.Is(err, bite.ErrExists):
        code = http.StatusConflict
    case errors.Is(err, bite.ErrNotExist):
        code = http.StatusNotFound
    default:
        code = http.StatusInternalServerError
    }
    http.Error(w, err.Error(), code)
}
```

We check errors and return relevant error text with an HTTP status code. `Error` responds with an error text (e.g., Not Found) and an HTTP status code (e.g., 404).

The `Is` function is the idiomatic way to compare errors. We should never directly compare errors since the error we're looking for could be buried elsewhere in the error chain. For instance, we could wrap two errors as follows and would miss the wrapped one:

```
// the err below wraps ErrInvalidRequest
err := fmt.Errorf("something bad happened: %w", bite.ErrInvalidRe
if err == bite.ErrInvalidRequest { .. } // false
```

This won't work because the error wraps `ErrInvalidRequest`. However, we could detect `ErrInvalidRequest` in the chain using the `Is` function (as we did in Listing 9.13).

9.5.4 Updating the server

We've integrated the storage into our handlers. It's time to update our server. As the next listing shows, we're adding a new parameter to `NewServer`, and then passing it to handlers. We don't hold on to `links` in the server since `links` is only about handlers, not the server.

Listing 9.17 Adding the Store to the server (link/server.go)

```
func NewServer(links *Store) *Server {

    mux := http.NewServeMux()
    mux.Handle("POST /shorten", Shorten(links))
    mux.Handle("GET /r/{key}", Resolve(links))
    mux.HandleFunc("GET /health", Health)
    return &Server{
        Handler: mux,
    }
}
```

That's all we need to do to update the server. The server gets a `Store` and passes it to handlers. Then, the handlers use that `Store` to create and retrieve links. As the next listing shows, we're done once we connect the store to the server.

Listing 9.18 Passing Store to the server (cmd/linkd/linkd.go)

```
func main() {
    ...

    links := link.NewServer(link.NewStore())

    ...
}
```

Now that we've updated all the components, it's time to run the server and try it out.

```
$ curl -i localhost:8080/r/nope

HTTP/1.1 404 Not Found
does not exist

$ curl -i localhost:8080/shorten -d 'key=go&url=https://go.dev'
HTTP/1.1 201 Created
go

$ curl -i localhost:8080/shorten -d 'key=go&url=https://go.dev'
HTTP/1.1 409 Conflict
already exists

$ curl -i localhost:8080/r/go
HTTP/1.1 302 Found
Location: https://go.dev
```

In this and the previous section, we looked at how the business logic and HTTP handlers collaborate to handle requests and responses. We also discussed the importance of using consistent data structures and shared errors. Isolating business logic and cross-cutting concerns from handlers improves our project's maintainability and simplicity.

9.6 Testing

Testing HTTP handlers involves sending them a `Request` and a

`ResponseWriter`, then watching what they output in their response. To monitor their responses, the `httptest` package provides a `ResponseWriter` called `ResponseRecorder`. For example, by passing a new `ResponseRecorder` to a handler, we can easily observe its response like this:

```
r := httptest.NewRequest(http.MethodGet, "/health", nil)    #A

w := httptest.NewRecorder()                              #B
Health(r, w)
```

Here, we send a new `Request` and `ResponseRecorder` to the `Health` handler. Later on, we can examine the handler's response through `w`. For example, we could log the handler's response status code and response body like this:

```
t.Log(w.Status)         // logs 200

t.Log(w.Body.String())  // logs OK     #A
```

To recap, we can send a `Request` and a `ResponseWriter`, like `ResponseRecorder`, to the handler, and then we can observe and verify its response matches what we expect.

Remember

`ResponseRecorder` implements the `ResponseWriter` interface. We can pass it to handlers to observe what they write to the response.

Dive deep: httptest.NewRequest vs http.NewRequest

Creating a new request using `httptest.NewRequest` is more convenient than using `http.NewRequest` because `httptest`'s approach is to panic instead of returning an error. So, we don't have to check for errors. However, it would be better if it were to take a `testing.TB` type. This way, we could have a failure report only for a specific test that calls `NewRequest`.

Luckily, we can do this ourselves. Below, we're creating a test helper to create a request.

```
func newRequest(tb testing.TB, method, target string, body io.Rea
    tb.Helper()
    r, err := http.NewRequest(method, target, body)
    if err != nil {
        tb.Fatal(err)
    }
    return r
}

func Test(t *testing.T) {
    r := newRequest(t, http.MethodGet, "/health", nil)
    // ... <- this code won't run if newRequest fails
}
```

On an error, instead of panicking, this helper generates a fatal error. This allows other tests to continue even after a fatal error (recall that panic stops the whole test run).

Testing handlers

As the next listing shows, as we did before, we pass a `Request` and a `ResponseRecorder` to the health handler. Then, we observe and verify the status code and body. Lastly, we test if the handler returns an OK status and an OK text.

Listing 9.19 Testing the Health handler (link/server_test.go)

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

    w := httptest.NewRecorder()
    r := httptest.NewRequest(http.MethodGet, "/", nil)

    Health(w, r)

    if w.Code != http.StatusOK {
        t.Errorf("got status code = %d, want %d", w.Code, http.St
    }
    if got := w.Body.String(); !strings.Contains(got, "OK") {
        t.Errorf("got body = %s\twant contains %s", got, "OK")
    }
}
```

Let's run the test.

```
$ go test
PASS
```

Testing the server

The previous test focused on the handler. Next, we're introducing something closer to an integration test to ensure the server directs requests to the right handler. As shown in the next listing, we're setting up a new server to request specific endpoints.

Listing 9.20 Testing server routing (link/server_test.go)

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

    tests := []struct {
        path     string
        method   string
        wantCode int
    }{
        {path: "/health", method: http.MethodGet, wantCode: http.
        {path: "/notfound", method: http.MethodGet, wantCode: htt

        /* exercise: add more test cases here for other routes */
    }
    for _, tt := range tests {
        t.Run(tt.path, func(t *testing.T) {
            t.Parallel()

            w := httptest.NewRecorder()
            r := httptest.NewRequest(tt.method, tt.path, nil)

            srv := NewServer(nil)                   #A
            srv.ServeHTTP(w, r)

            if w.Code != tt.wantCode {
                t.Errorf("got status code = %d, want %d", w.Code,
            }
        })
    }
}
```

This time, the request endpoint ("/health") matters because we're focusing on testing the server's ability to route, unlike the generic "/" endpoint used in

Listing 9.19. We've skipped comparing the response body (`w.Body.String()`) since our aim is just to check routing.

Let's run this test to see if it passes.

```
$ go test -run=TestServer
PASS
```

This confirms the server can route requests to the health handler. But without trying a missing route like "/notfound," we couldn't be sure the routing actually works—since without this test case, any request might just return an HTTP OK status by default.

Wrap up

- `ResponseRecorder` is a `ResponseWriter` that records handlers' responses.
- `httptest.NewRequest` returns a new `*Request` and panics on an error.
- We don't have to run a server to test the handlers.

9.7 Exercises

1. Modify `Shorten` to automatically generate keys for requests lacking a short link key.
2. Add an integer field to `Link`, incrementing with each `Retrieve` call.
3. Introduce a creation time field in `Link`, and set the time in `Create`.
4. Add support for updating an existing shortened URL's destination.
5. Develop a handler to delete a shortened URL using its short key.
6. Prevent infinite redirect loops by tracking redirects per client and URL.
7. Make `main` (`linkd`) testable by externalizing dependencies, as in Chapter 6.
8. Test the business logic (validation functions) and all `link` handlers.
9. Develop a client API for the link server, detailed in "`link/client.go`".
10. Write a CLI tool leveraging the client library in "`cmd/link`".
11. Compile and deploy the server across different operating systems.
12. Use the developed client and CLI to interact with the server locally.
13. Design an HTTP server API for the previous chapters' `hit` client.

Our server can shorten and resolve links using an in-memory database. In the next chapter, we'll dive into middleware and handler chaining patterns to improve our project further.

9.8 Summary

- Circular import is a compile-time error when Package A imports B and B imports A. The layered package structure makes this issue less likely.
- The `http` package provides `Server` to serve incoming requests with a `Handler`.
- Handler takes a *Request and a ResponseWriter.
- `ResponseWriter` is an interface with `Header`, `Write`, and `WriteHeader` methods.
- Timeouts can be set up to protect servers from malicious clients who may try to abuse the system with extended open connections or excessive data.
- `ServeMux` is a handler that routes requests to handlers based on route patterns. It supports versatile routing patterns like wildcard segments.
- Business logic provides consistency and handles business rules to maintain a clean and maintainable codebase, while handlers manage HTTP requests and responses.

10 Functional Programming with Middleware

This chapter covers

- Middleware patterns to enhance HTTP handlers with cross-cutting functionalities.
- Handler chaining pattern and functional programming techniques to reduce redundancy and streamline the codebase.

The last chapter explored writing an HTTP server with a layered package architecture and idiomatic naming conventions. This chapter will explore functional programming and middleware and handler chaining patterns. We'll also explore JSON. Along the way, we'll add tracing support to our project using the `context` and `slog` packages.

Note

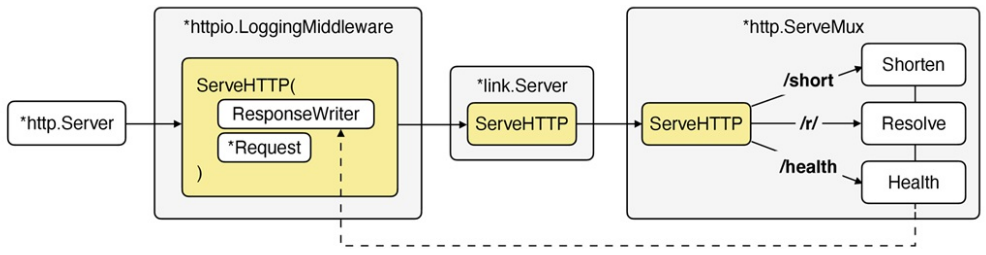
Find the source code for this chapter at https://github.com/inancgumus/gobyexample/tree/main/bite.

10.1 Middleware

Middleware is a reusable handler that wraps another handler. We frequently use middleware to provide cross-cutting concerns like authentication, data compression, and logging.

Imagine Lina, the product owner in our team, wants us to log every HTTP handler request so that she can quickly pinpoint problems and gain insights into user behavior. We can do what she wants by integrating a *logging middleware* into our server. As Figure 10.1 shows, the middleware wraps the server's handler, monitors, and logs every request and response.

Figure 10.1 The logging middleware monitors all requests and responses.



Handlers won't know that the middleware is monitoring every request. The beauty of middleware is that we can integrate it into the handler chain without modifying the handlers.

10.1.1 Logging middleware

We'll declare the middleware in `httpio`. This package provides cross-cutting concerns to handlers. So, future code can import and use the same middleware. See the next listing.

Listing 10.1 Writing a logging middleware (httpio/middleware.go)

```
package httpio

import (
    "log/slog"
    "net/http"
    "time"
)

// LoggingMiddleware logs requests.
func LoggingMiddleware(next http.Handler) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        start := time.Now()

        next.ServeHTTP(w, r)                   #B

        slog.Log(r.Context(), slog.LevelInfo,
            "request",                        #C
            "url", r.URL, "method", r.Method,            #D
            "took", time.Since(start))                #D
    }
}
```

`LoggingMiddleware` is a higher-order function that takes a handler and returns another handler. It wraps the handler and measures how long it takes the wrapped handler to process the request. For each incoming request, it records the time, calls the wrapped handler, and logs the request duration and details about the incoming request. But it doesn't write anything to the client. Instead, it delegates the request processing and response to the wrapped handler. The middleware only monitors and logs requests.

Integration

Since we want to log every incoming request, we'll integrate the middleware into our server's handler. This way, our middleware can monitor and log every incoming request. As the next listing shows, once we wrap `handler`, the rest of the routing will be automatic. For instance:

```
http.Server -> LoggingMiddleware -> TimeoutHandler -> link.Server
```

Each handler above calls each other's `ServeHTTP` method to run this chain. `ResponseWriter` and `*Request` emanates and goes from `http.Server` until the last handler in the chain.

Once the HTTP server gets a request, it redirects the request to its handler, which is our middleware. The middleware calls the timeout handler, which calls the link server. Lastly, the link server routes the request to one of its handlers.

Listing 10.2 Wrapping the server's handler (cmd/linkd/linkd.go)

```
func main() {
    ...

    links   := link.NewServer(link.NewStore())
    handler := http.TimeoutHandler(links, timeout, "timeout")
    handler  = httpio.LoggingMiddleware(handler)            #A

    srv := &http.Server{
        Addr:        addr,
        Handler:     handler,
        ReadTimeout: timeout,
    }
    ...
}
```

This integration was straightforward. We've added logging support to the server without changing the handlers' code. We can now see the logs. Let's send our server a few requests.

```
$ go run ./cmd/linkd

INFO starting app=linkd addr=localhost:8080
INFO request app=linkd url=/shorten method=POST took=245.292µs
INFO request app=linkd url=/r/go method=GET took=36.709µs
INFO request app=linkd url=/health method=GET took=21.791µs
```

Now, let's explore how to enrich the logger middleware with more information.

10.1.2 Capturing status codes

Lina was pleased with how quickly we added logging and now she's asking for more: logging HTTP status codes like 200 (OK). Our middleware, as it stands, can't directly capture these status codes because they're set within the handlers using the `ResponseWriter` interface:

```
type ResponseWriter interface {
    Header() Header
    Write([]byte) (int, error)
    WriteHeader(statusCode int)    #A
}
```

For instance, a handler can write a status code calling `WriteHeader` like this:

```
func(w http.ResponseWriter, r *http.Request) {

    w.WriteHeader(http.StatusOK)
}
```

Luckily, we can define our own `responseRecorder` type that implements this interface by embedding a `ResponseWriter` and an additional field to hold the status code. When the handler calls `WriteHeader` to set the status code, our `responseRecorder` will record it.

See the next listing.

Listing 10.3 Capturing status codes (httpio/middleware.go)

```
type responseRecorder struct {
    http.ResponseWriter                        #A
    status int
}

func (r *responseRecorder) WriteHeader(code int) {
    r.status = code                        #B
    r.ResponseWriter.WriteHeader(code)                #C
}

func LoggingMiddleware(next http.Handler) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {    #D
        start := time.Now()

        rr := &responseRecorder{                   #E
            ResponseWriter: w,                 #E
        }                              #E
        next.ServeHTTP(rr, r)                    #F


        slog.Log(r.Context(), slog.LevelInfo,
            "request",
            "url", r.URL, "method", r.Method,
            "status", rr.status,                   #G
            "took", time.Since(start))
    }
}
```

When we wrap a handler with this middleware, we pass in our `responseRecorder`. The handler does its thing while we capture the status code for logging. By augmenting `ResponseWriter` this way, we can now log status codes. Let's give it a try.

```
$ go run ./cmd/linkd

INFO starting app=linkd addr=localhost:8080
INFO request app=linkd url=/shorten method=POST status=201 took=2
INFO request app=linkd url=/r/go method=GET status=302 took=56.41
INFO request app=linkd url=/r/nope method=GET status=404 took=26.
INFO request app=linkd url=/health method=GET status=200 took=29.
```

Note

As discussed before, embedding an interface doesn't come with an implementation. Luckily, we're wrapping the underlying `ResponseWriter` provided by `http.Server`.

10.1.3 Testing the middleware

Now that we have the logging middleware, it's time to test to verify if it works as intended. As shown in the next listing, we're verifying what it logs into an in-memory buffer.

Listing 10.4 Testing LoggingMiddleware (httpio/middleware_test.go)

```
package httpio

import (
    "bytes"
    "log/slog"
    "net/http"
    "net/http/httptest"
    "strings"
    "testing"
)

func TestLoggingMiddleware(t *testing.T) {
    var buf bytes.Buffer                    #A
    log := slog.New(slog.NewTextHandler(&buf, nil))        #A
    slog.SetDefault(log)                    #A

    var handler http.Handler
    handler = http.HandlerFunc(func(w http.ResponseWriter, _ *htt
        w.WriteHeader(418)
    })
    handler = LoggingMiddleware(handler)            #B
    handler.ServeHTTP(                     #B
        httptest.NewRecorder(),                  #B
        httptest.NewRequest("GET", "/test", nil),        #B
    )                                 #B

    got := buf.String()                    #C
    if !strings.Contains(got, "GET") {            #C
        t.Error("want GET in the log")            #C
    }                                 #C
    if !strings.Contains(got, "418") {             #C
        t.Error("want 418 in the log")             #C
    }                            #C
    if !strings.Contains(got, "/test") {             #C
        t.Errorf("want /test in the log")            #C
    }                            #C
    if t.Failed() {                    #D
        t.Log("got:", got)                #D
    }                            #D
}
```

The `slog` handlers take an `io.Writer` as their first parameter. So, as we did in the previous chapters, we can pass a `bytes.Buffer` to this handler to observe the logger.

```
func NewTextHandler(w io.Writer, opts *HandlerOptions) *TextHandl
```

Once we set the logger up, we craft an HTTP handler that responds with a random HTTP status code like 418. After that, we wrap the handler with the middleware. Lastly, we fetch the logger's output from the buffer and verify if the middleware logs correct values.

`Failed` at the bottom is for dumping all the logs once if the current test fails. If we printed the logs after each `Error`, the test logs would be cluttered with duplicate logs.

Running the test

We've set up a buffer to observe what the middleware logs when a handler is called. Once we run the middleware, it should call the wrapped handler and then log a message with the request method ("GET"), status code ("418"), and the path ("/test"). Let's test it.

```
$ go test ./httpio -run=TestLoggingMiddleware
ok
```

As an exercise, try breaking the logger and see how the test fails. For instance, the test would log the following if the status code didn't match:

```
$ go test ./httpio -run=TestLoggingMiddleware
--- FAIL: TestLoggingMiddleware
```

```
    want 418 in the log
    got: level=INFO msg=request url=/test method=GET status=200 t
```

10.1.4 Optional (obscure) interfaces

While our logging middleware effectively records status codes from handlers, it may hide *optional interfaces* implemented by the `ResponseWriter`'s underlying type, such as:

- `http.Flusher`: Enables sending buffered response data to clients immediately.
- `http.Pusher`: Allows for HTTP/2 server push.
- `http.Hijacker`: Enables seizing control of the underlying TCP connection (e.g., so that the handler can upgrade the connection to a WebSocket connection).

For instance, the `Flusher` interface has a `Flush` method. Since `ResponseWriter` doesn't have a `Flush` method, a handler should explicitly check for `Flusher` using a type assertion:

```
func(w http.ResponseWriter, r *http.Request) {
    if f, ok := w.(http.Flusher); ok {    #A
        f.Flush()                #B
    }
}
```

Note

Remember type assertions from "Section 9.3.1's Dive deep: Be careful when embedding a type". It also discusses the following embedding problem and explains why it happens.

If we wrap the handler above with our logging middleware, that handler won't see the `Flush` method. But why? Go loves explicitness and simplicity. When our middleware embeds `ResponseWriter` (say with a `Flusher` underneath) as follows, only the `ResponseWriter`'s methods are promoted to `responseRecorder` (not the methods of the type underneath).

```
type responseRecorder struct {
    http.ResponseWriter        #A
    status int
}
```

The handler above will get a `ResponseWriter` with a `responseRecorder` underneath. When it checks if there is a `Flusher` under it, Go would say, "No, there is not!". Luckily, we can expose the underlying `ResponseWriter`. See the next listing.

Listing 10.5 Implementing Unwrap (httpio/middleware.go)

```
type responseRecorder struct {
    http.ResponseWriter
    status int
}

func (r *responseRecorder) Unwrap() http.ResponseWriter {    #A
    return r.ResponseWriter                              #A
}                            #A
```

Then, we must update the handler to use a type called `ResponseController` to let the handler use the optional interfaces. See the example below. Calling `Flush` flushes the buffer if `Flusher` is supported. Otherwise, it returns `ErrNotSupported`.

```
func(w http.ResponseWriter, r *http.Request) {
    rc := http.NewResponseController(w)
    if err := rc.Flush(); errors.Is(err, http.ErrNotSupported) {
        /* sorry, ResponseWriter does not support Flusher */
    }
}
```

Underneath, calling `Flush` causes `ResponseController` to call our middleware's `Unwrap` method to extract the underlying `ResponseWriter` it exposed, check to see `Flusher` is supported, and call `Flush`. In summary, if a middleware wraps a `ResponseWriter`, it should implement `Unwrap`. Handlers should use `ResponseController` to access optional interfaces.

`ResponseController` also lets handlers set their own read and write timeouts (recall from "Section 10.1.3 Hardening the server" that we set hard-coded timeouts on the server).

```
func(w http.ResponseWriter, r *http.Request) {
    rc := http.NewResponseController(w)
    err := rc.SetReadDeadline(time.Now().Add(time.Minute)) // ...
}
```

This won't be effective if the server's `ReadTimeout` is below one minute. We should omit or set the server timeouts to larger values to adjust granular timeouts inside handlers. For more information about `ResponseController`, visit: pkg.go.dev/net/http#ResponseController

Exercise

Write a test to verify `LoggingMiddleware` doesn't hide optional interfaces.

Dive deep: Asserting to an anonymous interface

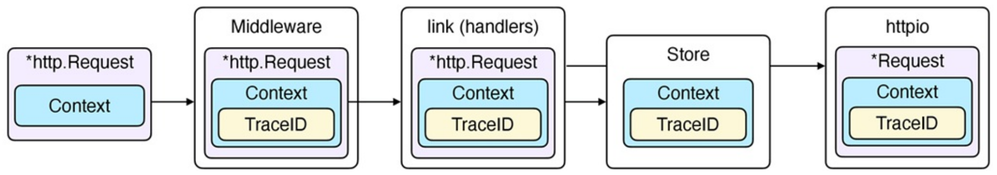
Type asserting an interface using an anonymous type is also possible. For example, this asserts for `Flush`: `w.(interface{ Flush() })`. This will return a type with a `Flush` method if `w`'s underlying type has a `Flush` method. This becomes useful since it saves us from declaring a type. Since `Flusher` already exists, we didn't do that in the previous example.

10.1.5 Trace ID and context values

Although we have a good enough logging system, we don't have a way to match logs to HTTP requests. Adding a "Trace ID" to our log entries would enable us to group logs by requests, simplifying diagnosing what exactly is happening with a request's lifecycle. For instance, an administrator can find the matching log entries related to a single request.

Besides propagating cancellation signals, `Context` can also transmit values. See Figure 10.2. We can transmit a request's context from a middleware to the rest of the system. This means we can relay the trace ID throughout our system and use the ID while logging.

Figure 10.2 The middleware adds "TraceID" to the request's context. The TraceID is relayed to the remaining parts of the call chain through the request context.



You might have noticed that we've been relaying the request's context while logging:

```
slog.Log(r.Context(), slog.LevelInfo, ..)    #A
```

Calling `Log` (or other logger methods) delegates the handling of a log record (includes log attributes like "app=linkd", "method=GET", and so on) to a type that satisfies the `slog.Handler` interface. So, we can craft a custom `slog.Handler` that can extract the trace ID put in the context and log with that ID. Once we make a logger with our custom handler, every logger method can automatically produce a log line with trace IDs.

To wrap up, we can inject a trace ID from middleware to each request's context and then set a custom logger to extract and log with these IDs from the context automatically.

Tracing middleware

We'll inject a random trace ID into the context within our new `TraceMiddleware`. It's a good practice to store a value with a unique key type within a context to prevent clashes with other potential values. So, we'll create a new type for storing a trace ID within the context.

See the next listing.

Listing 10.6 Implementing TraceMiddleware (httpio/middleware.go)

```
type traceIDKey struct{}                        #A


func TraceMiddleware(next http.Handler) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        ctx := SetTraceID(                        #B
            r.Context(),                          #B
            rand.Int64(),                          #C
        )
        next.ServeHTTP(w, r.WithContext(ctx))            #D
    }
}

func SetTraceID(ctx context.Context, id int64) context.Context {
    return context.WithValue(                   #F
        ctx,                             #F
        traceIDKey{},                         #G
        id,                              #H
    )
}

func TraceID(ctx context.Context) (int64, bool) {            #I
    n, ok := ctx.Value(traceIDKey{}).(int64)            #J
    return n, ok
}
```

We're declaring a new type for the context key, crafting a middleware to add the trace ID to the request's context, and forging a new request with the new context. `TraceMiddleware` is the middleware function. `SetTraceID` and `TraceID` are helpers. `SetTraceID` derives and returns a new context with trace ID, while `TraceID` returns the trace ID from the context.

Let's go over this code in more detail.

Explaining the code

In `TraceMiddleware`, we generate a pseudo-random 64-bit integer as the trace ID for each request. Although this is fine for our case, using UUID in non-trivial programs to prevent conflicts is better. Then, we're deriving a new context using `SetTraceID` with a random ID. Lastly, we're running the wrapped handler with a new request and context. This way, the wrapped handler (or the logger) can get the trace ID from the context.

Note

UUID means a universally unique ID. See pkg.go.dev/github.com/google/uuid

Let's look at the helper functions.

In `SetTraceID`, using `WithValue`, we derive a new context from an existing context.

```
func WithValue(parent Context, key any, val any) Context
```

`WithValue` takes a key and a value as the `any` type, an empty interface. An interface carries both type information and value, so creating a new type in our own package avoids clashes with the context keys from other packages. Also, an empty struct as a context key is efficient—it saves on memory, which matters since we're generating a new context for each HTTP request. In short, `traceIDKey` is unique to our package, so it avoids key collisions.

Tip

Use an empty struct as a context value key type to avoid collisions and for efficiency. Also, only inject request-scoped data into a context. Using it as a general-purpose data store for data can lead to code that is harder to understand, maintain, and debug.

Lastly, in `TraceID`, we extract the trace ID from the context using the `Value` method:

```
id, ok := ctx.Value(traceIDKey{}).(int64)    #A

              ^                       ^
    extracts the context value  is it int64?
```

Recall that a context stores a value in an `any` interface. Here, we assert that the stored value is a 64-bit integer. Since `SetTraceID` saves the value as `int64`, `ok` will be `true`. And, `id` will be `int64` and the value we inject to the context from the tracing middleware.

Tip

Helper functions streamline setting and getting context values consistently. They're useful because context values can be of any type and keys.

Testing the trace middleware

Before writing a slog handler to log trace IDs, let's test the trace middleware.

See the next listing.

Listing 10.7 Testing TraceMiddleware (httpio/middleware_test.go)

```
func TestTraceMiddleware(t *testing.T) {
    var (
        traceID int64
        ok      bool
    )

    var handler http.Handler
    handler = http.HandlerFunc(func(_ http.ResponseWriter, r *htt
        traceID, ok = TraceID(r.Context())
    })
    handler = TraceMiddleware(handler)

    handler.ServeHTTP(
        httptest.NewRecorder(),
        httptest.NewRequest("GET", "/test", nil),
    )
    if !ok {
        t.Fatal("got context without a trace id")
    }
    if traceID <= 0 {
        t.Fatalf("got %d, want a positive trace id", traceID)
    }

    prev := traceID
    handler.ServeHTTP(
        httptest.NewRecorder(),
        httptest.NewRequest("GET", "/test", nil),
    )
    if prev == traceID {
        t.Fatalf("got duplicate trace id: %d", traceID)
    }
}
```

We're getting the trace ID from the request's context in the handler. Then, we're wrapping and running that handler with the trace middleware. After that, we're verifying if the handler gets a positive trace ID. This is good enough to test that the middleware works.

Lastly, we're rerunning the tracer to verify that we get different trace IDs. This approach is more maintainable compared to checking exact trace IDs. Otherwise, the test would be complicated and we would tie the trace ID generation logic to the test.

We've written a test to verify that the handlers can get different trace IDs.

Let's run the test.

```
$ go test ./httpio -run=TestTraceMiddleware
ok
```

Tip

Avoid binding the implementation logic to tests for better maintenance.

Writing a slog handler

The next step is to write a `slog` handler that will be called each time a `slog` method is invoked (e.g., `slog.Log(..)`). For that, we'll implement the `slog.Handler` interface:

```
type Handler interface {
    Enabled(context.Context, Level) bool    #A
    Handle(context.Context, Record) error    #B
    WithAttrs(attrs []Attr) Handler    #C
    WithGroup(name string) Handler        #D
}
```

The next listing shows our new type that implements the `Handler` interface. `LogHandler` is a wrapper around an existing `Handler` that augments logs with a trace ID (that's why we embed `Handler`). This frees us from crafting a full-fledged handler, like `TextHandler` that prints logs as text (there is also an existing `JSONHandler` that prints logs as JSON).

Listing 10.8 Implementing LogHandler (httpio/middleware.go)

```
type LogHandler struct{ slog.Handler }                    #A
```

```
func (h *LogHandler) Handle(ctx context.Context, r slog.Record) e
    if id, ok := TraceID(ctx); ok {                    #C
        r.Add("trace_id", id)                        #D
    }
    return h.Handler.Handle(ctx, r)                    #E
}

func (h *LogHandler) WithAttrs(attrs []slog.Attr) slog.Handler {
    return &LogHandler{Handler: h.Handler.WithAttrs(attrs)}
}

func (h *LogHandler) WithGroup(name string) slog.Handler {
    return &LogHandler{Handler: h.Handler.WithGroup(name)}
}
```

`Handle` extracts the trace ID, augments the log record with a `trace_id` key, and calls the underlying handler. Once we set up a logger that uses `LogHandler`, every time we call a log method, it'll output a log line with a trace ID if it exists in the context. For instance:

```
// in main():
log := slog.New(&httpio.LogHandler{                #A
    Handler: slog.NewTextHandler(os.Stderr, nil),            #B
})
// ...
log.Log(ctx, slog.LevelInfo, "request", "method", "GET")
// logs: request method=GET trace_id=43838438934
// (assuming the context has a trace ID value)
```

We've also implemented `WithAttrs` and `WithGroup` methods to keep processing the logs with `LogHandler` in case we derive a new logger using these methods. For instance, the following derives a new logger that always outputs with "app=linkd":

```
// following from the code snippet above
log = log.With("app", "linkd")                    #A
log.Log(ctx, slog.LevelInfo, "request", "method", "GET")
// logs: app=linkd request method=GET trace_id=43838438934
```

Implementing `WithAttr` and returning a new `LogHandler` handler that wraps the underlying handler allows us to derive a new logger that will use the `LogHandler`. This way, the derived logger can still extract the trace ID each time a logger method is called.

Testing the slog handler

Before wiring up the trace middleware and the `slog` handler, let's test the `slog` handler.

See the next listing. We test if `LogHandler` can output logs with trace IDs.

Listing 10.9 Testing LogHandler (httpio/middleware_test.go)

```
func TestLogHandler(t *testing.T) {
    var buf bytes.Buffer

    ctx := SetTraceID(context.Background(), 42)        #A
    log := slog.New(&LogHandler{                #B
        Handler: slog.NewTextHandler(&buf, nil),        #B
    })                            #B

    log.Log(ctx, slog.LevelInfo, "test")            #C

    if got := buf.String(); !strings.Contains(got, "42") {    #D
        t.Errorf("want trace id %d in the log:%s", 42, got)
    }
}
```

As usual, we're creating an in-memory `bytes.Buffer` to observe the logger. Then, we're injecting the trace ID into the context using `SetTraceID`. After that, we're creating a `slog` logger using `LogHandler` (that wraps the existing `TextHandler`).

The next step is to log something and check the log line contains the trace ID. This approach is maintainable because we're testing *the actual behavior* instead of using `SetTraceID` to set the trace ID into the context and then using `TraceID` to get back the trace ID.

We've written a test to verify that the slog handler can set and get trace IDs. Let's try.

```
$ go test ./httpio -run=TestLogHandler
ok
```

Tip

Also use `slogtest` to test handlers for compatibility. See pkg.go.dev/testing/slogtest. Since our handler is a wrapper around an existing handler, we didn't use the `slogtest` package.

Wiring the parts

Now that we can inject a trace ID using `TraceMiddleware` and augment logs with a trace ID using `LogHandler`, let's wire these up in the `main` function. See the next listing.

Listing 10.10 Wiring up the tracer (cmd/linkd/linkd.go)

```
func main() {
    ...

    log := slog.New(&httpio.LogHandler{            #A
        Handler: slog.NewTextHandler(os.Stderr, nil),    #B
    })
    log = log.With("app", "linkd")            #C
    slog.SetDefault(log)

    log.Info("starting the server", "addr", addr)

    links   := link.NewServer(link.NewStore())
    handler := http.TimeoutHandler(links, timeout, "timeout")
    handler = httpio.LoggingMiddleware(handler)
    handler = httpio.TraceMiddleware(handler)        #D

    srv := &http.Server{
        Addr:        addr,
        Handler:     handler,
        ReadTimeout: timeout,
    }
    if err := srv.ListenAndServe(); !errors.Is(err, http.ErrServe
        log.Error("server closed unexpectedly", "message", err)
    }
}
```

We've created a new logger that will handle logs with `LogHandler`. The logger will output logs with the text format as the underlying handler is the text log handler (`TextHandler`).

Using `With`, we derive a new logger that will include "app" and "linkd" on each log output. Underneath, calling `With` calls `LogHandler.WithAttr` to derive a new logger with the same `LogHandler`. This way, the derived logger can still output trace IDs. If we hadn't implemented `WithAttr` in `LogHandler`, the derived logger would use the embedded `TextHandler`'s `WithAttr` method, returning a `TextHandler` instead of `LogHandler`.

Lastly, we wrap the logging middleware with the tracing middleware to inject trace IDs. With these changes, calling the log methods will output trace IDs. We can now correlate requests using trace IDs. For instance, the following first log line is from the logging middleware and the next is from `Store.Retrieve`. The last log line is from the logging middleware.

```
$ go run ./cmd/linkd


level=INFO msg=request app=linkd url=/r/fortesting method=GET sta
level=ERROR msg=internal app=linkd url=/r/fortesting message="int

time=2024-02-29T15:11:12.495+03:00 level=INFO msg=request app=lin
```

The first two logs have the same trace ID, while the last one is different. Each request will have a different trace ID, but the same request will always have the same ID. This way, we can see how our program handles a request and what happens while doing so in between.

10.1.6 Wrap up

- Middleware is a handler that takes and returns a handler.
- Wrapping `ResponseWriter` allows us to capture handler responses.
- Embedded interfaces can hide the capabilities of the underlying types.
- `ResponseController` allows using optional interfaces without a type assertion.
- `ResponseController` lets handlers set their own read and write timeouts.
- `Unwrap` returns the underlying `ResponseWriter` (if `Unwrap` is available).
- `Context.WithValue` attaches a value to a `Context` derived from an existing one.
- `Context.Value` extracts a value from a `Context` using a key.
- `Context` keys should be unique types to prevent key collisions.

- `slog` loggers handle log records with a type that implements `slog.Handler`.
- Like `WithGroup`, `WithAttr` derives a new `Handler` from an existing one.

10.2 Handler chaining pattern

Consider the following code:

```
handler := func(w http.ResponseWriter, r *http.Request) {
    // ...
    link, err := links.Retrieve("non-existing key")
    if err != nil {
        httpError(w, err)
        return                     #A
    }
    http.Redirect(w, r, link.URL, http.StatusFound)
}
```

This handler should *return after* an error occurs to prevent any wonky behavior. Otherwise, there can be unexpected results and even security vulnerabilities. It's fair to ask, "But how can we avoid adding a return statement every time?" Remember that Go is a compiled language. If we carefully arrange our code, we can get the compiler's help if a handler forgets to return (it would be a compile-time error rather than runtime (=better)).

Avoid forgetting to return

The trick is to create a new handler type that returns another. This function returns a handler to continue processing the request (e.g., when an error occurs) or returns `nil` for success.

```
handler := func(w http.ResponseWriter, r *http.Request) Handler {

    // ...
    if err != nil {
        return httpError(..)            #A
    }
    http.Redirect(w, r, link.URL, http.StatusFound)
    return nil                     #B
}
```

```
http.HandlerFunc(handler) // won't work now!
```

This function guarantees a return, ensuring it's impossible to overlook. But its signature doesn't conform to the typical `Handler` interface. We can't convert it using `HandlerFunc`.

httpio.Handler

To solve the previous issue, we introduce a new handler type in the next listing. This type is like `HandlerFunc` with a difference: it calls the next handler after calling the wrapped handler, creating a handler chain that goes forever until one of them returns `nil` (success).

Listing 10.11 Declaring httpio.Handler (httpio/httpio.go)

```
package httpio

import "net/http"

type Handler func(w http.ResponseWriter, r *http.Request) Handler

func (h Handler) ServeHTTP(w http.ResponseWriter, r *http.Request
    if next := h(w, r); next != nil {                #C
        next.ServeHTTP(w, r)                     #C
    }
}
```

For instance, we can convert the previous function to an `httpio.Handler`. Since, `httpio.Handler` has a `ServeHTTP` method, it implements the `http.Handler` interface.

```
ListenAndServe("localhost:8080", httpio.Handler(handler)) // ok
```

We'll make all the handlers the `httpio` handlers. Like middleware, they'll return a handler to keep processing the request. Once the last one returns `nil`, the chain will stop. Before that, let's declare some `httpio` convenience handlers for managing common handler work.

10.2.1 Adding helper handlers

Instead of returning an error and status code using `http.Error`, we'll use our own functions compatible with the `http` and `httpio` packages. These helpers in the next listing should look familiar to those experienced in functional programming. Each helper can be a part of the handler chain and responsible for only one thing. For example:

- `OK` returns a `nil` handler to stop the chain. Useful for readability.
- `Code` writes a status code to the response and returns the given handler.
- `Text` writes a text message to the response. Then, it returns `OK` to halt the chain.

It's important to make each one focused. This way, we can chain them in creative ways. Also, we could add more helpers, but these are enough for us to restructure the server.

Listing 10.12 Declaring helper httpio handlers (httpio/httpio.go)

```
package httpio

// ..

func OK(w http.ResponseWriter, r *http.Request) Handler {
    return nil
}

func Code(statusCode int, next Handler) Handler {            #B
    return func(w http.ResponseWriter, r *http.Request) Handler {
        w.WriteHeader(statusCode)                        #C
        return next                            #D
    }
}

func Text(s string) Handler {                        #E
    return func(w http.ResponseWriter, r *http.Request) Handler {
        w.Header().Set("Content-Type", "text/plain; charset=utf-8
        fmt.Fprintln(w, s)                        #F
        return OK                            #F
    }
}
```

To understand how this code works, let's remember the `ServeHTTP` method from Listing 10.11. This function runs the handlers until it sees a `nil` handler (like the `OK` handler).

```
func (h Handler) ServeHTTP(w http.ResponseWriter, r *http.Request
    if next := h(w, r); next != nil {
        next.ServeHTTP(w, r)
    }
}
```

For example, we can chain the handlers above to write an OK status with a message:

```
func(w http.ResponseWriter, r *http.Request) httpio.Handler {

    // ...
    return httpio.Code(http.StatusOK, httpio.Text("everything is
}
```

Underneath, `Code` writes the OK status code using `WriteHeader`, and then returns the `Text` handler. `ServeHTTP` detects that the chain is active, and runs `Text`. Inside `Text`, it writes the message to the response, returns `OK`, and stops the chain. The client will see this:

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8

everything is great!
```

Another benefit of returning handlers is that each can access the current `ResponseWriter` and `Request` (unlike `http.Error`, which expects a `ResponseWriter`). This will allow us to do many interesting things (e.g., detecting if the status code is an internal error and log).

Exercise

Add a `CodeText` handler that responds with a status code and a text. We should be able to use it like this: `httpio.CodeText(http.StatusOK, "everything is great!")`

10.2.2 Converting HTTP errors

Before jumping into restructuring the server handlers, let's first restructure the `httpError` function (which converts business logic errors to HTTP codes). Currently, it takes a `ResponseWriter` and calls `http.Error` to report the error with a status code.

```
func httpError(w http.ResponseWriter, err error) {

    // ...
    http.Error(w, err.Error(), code)
}
```

We'll move this function to the `httpio` package as it becomes generic enough to be used by all the servers in our company (although we have only one server). See the next listing.

Listing 10.13 Moving httpError to httpio.Error (httpio/httpio.go)

```
func Error(err error) Handler {                #A

    if err == nil { // no error
        return OK                        #B
    }
    var code int
    switch {
    case errors.Is(err, bite.ErrInvalidRequest):
        code = http.StatusBadRequest
    case errors.Is(err, bite.ErrExists):
        code = http.StatusConflict
    case errors.Is(err, bite.ErrNotExist):
        code = http.StatusNotFound
    default:
        code = http.StatusInternalServerError
    }
    return Code(code, Text(err.Error()))            #C
}
```

We've renamed the function to `Error` and moved it into `httpio`. The biggest difference is that now it's an `httpio` handler and chainable like the others. The handlers can now use `Error` to report errors instead of using `Code` and `Text` helpers. Still, we'll keep them around. This gives handlers the freedom to choose which one to use. This is one of the benefits of the handler chaining pattern. It doesn't take freedom away from handlers.

10.2.3 Restructuring handlers

Now that we have `httpio` handlers and restructured `httpError`, it's time to restructure the server handlers. See the next listing. We're making the handlers return an `httpio` handler. They're now calling `httpio.Error` to report errors and `OK` to stop the chain.

Listing 10.14 Restructuring server handlers (link/server.go)

```
func Shorten(links *Store) httpio.Handler {                #A

    return func(w http.ResponseWriter, r *http.Request) httpio.Ha
        link := Link{
            Key: r.FormValue("key"),
            URL: r.FormValue("url"),
        }
        if err := links.Create(r.Context(), link); err != nil {
            return httpio.Error(err)                    #B
        }

        return httpio.Code(http.StatusCreated, httpio.Text(link.K
    }
}

func Resolve(links *Store) httpio.Handler {
    return func(w http.ResponseWriter, r *http.Request) httpio.Ha
        link, err := links.Retrieve(r.Context(), r.PathValue("key
        if err != nil {
            return httpio.Error(err)
        }

        http.Redirect(w, r, link.URL, http.StatusFound)

        return httpio.OK                            #D
    }
}
```

Let's give it a try.

```
$ curl -i localhost:8080/shorten -d 'key=go&url=https://go.dev'

HTTP/1.1 201 Created
go
$ curl -i localhost:8080/r/go
```

```
HTTP/1.1 302 Found
<a href="https://go.dev">Found</a>.
$ curl -i localhost:8080/r/nope
HTTP/1.1 404 Not Found
does not exist
```

The behavior is the same as before. The handler-chaining approach reduces repetitive code and ensures handlers remember to return. By structuring code this way, the compiler will force us to write more reliable code, making surprises or security vulnerabilities less likely.

10.2.4 Leaky internal errors

Lina from the previous section noticed that the server sometimes responds with an internal status code and shows the actual internal error:

```
$ curl -i localhost:8080/r/fortesting

HTTP/1.1 500 Internal Server Error
internal error: ...db at IP ... failed    #A
```

This makes her uncomfortable as she thinks the system's internals shouldn't be exposed. Instead, she wants us to hide the internal errors from the clients and still log them so we can diagnose and fix the potential issues. To address Lina's concern, the next listing updates `Error` to distinguish internal errors. If an internal error happens, `Error` logs it and returns a generic error message to the client, keeping the complete error details in the server logs.

Listing 10.15 Logging and hiding internal errors (httpio/httpio.go)

```
func Error(err error) Handler {
    ...

    return func(w http.ResponseWriter, r *http.Request) Handler {

        if code == http.StatusInternalServerError {           #B
            slog.Log(r.Context(), slog.LevelError,           #B
                "internal", "url", r.URL, "message", err)
            err = bite.ErrInternal                          #B
        }                           #B

        return Code(code, Text(err.Error()))
    }
}
```

Instead of directly returning from `Code`, we've chained another handler in between. This allowed us to extract the request's `Context` and URL. This wouldn't be possible without the flexibility the handler chaining pattern provides. Let's give it a try.

```
$ curl -i localhost:8080/r/fortesting
```

```
HTTP/1.1 500 Internal Server Error
internal error: please try again later or contact support
```

And here's the server's log in the meantime:

```
2061/07/28 15:48:13 ERROR internal app=linkd url=/r/fortesting me

2061/07/28 15:48:13 INFO request app=linkd url=/r/fortesting meth
```

The client receives a generic internal error message while the server logs show the actual internal error details. With these changes, Lina will be much happier.

10.2.5 Wrap up

- Not returning from handlers when required can result in unpredictable outcomes.
- Using the compiler and type system's power leads to resilient code and systems.
- The handler chaining pattern flexibly chains handlers to serve a request.

10.3 JSON

As the Bite team is getting closer to completing the server, Lina wanted us to add JSON support so that other tools in the company can integrate with the server.

JSON (JavaScript Object Notation) is a lightweight format for storing and transmitting data, often used for exchanging information between a web server and a client. It's easy to read and write for humans, and simple to parse and generate for machines. In this section, we'll add JSON encoding and decoding support to the server using Go's `json` package.

This way, other tools (e.g., HTTP clients) can talk to the server using JSON.

Note

For more information, visit: pkg.go.dev/encoding/json and go.dev/blog/json.

10.3.1 Encoding and decoding helpers

The `json` package has an `Encoder` type to encode a value to JSON, and a `Decoder` type to decode JSON into a value (i.e., a variable). As the next listing shows, using these types, we're adding two helper functions to the `httpio` package to encode and decode JSON:

- `JSON` encodes and writes a value as JSON to the client.
- `DecodeJSON` decodes JSON into `v`. It makes sure the result matches the underlying type of `v` (e.g., for a struct pointer, JSON keys must match the struct's fields).

While `JSON` returns a handler, `DecodeJSON` returns an error. This is because we'll use `JSON` in the handler chain. The latter is not a handler but only for receiving and decoding JSON.

Listing 10.16 Adding JSON helpers (httpio/json.go)

```
package httpio

import (
    "encoding/json"
    ...
)

func JSON(v any) Handler {                       #A
    return func(w http.ResponseWriter, r *http.Request) Handler {
        w.Header().Set("Content-Type",                 #B
            "application/json; charset=utf-8")           #B
        if err := json.NewEncoder(w).Encode(v); err != nil {
            slog.Log(r.Context(), slog.LevelError,            #D
                "internal", "url", r.URL, "message", err)
        }

        return OK
    }
}

func DecodeJSON(r io.Reader, v any) error {
    decoder := json.NewDecoder(r)                   #E
    decoder.DisallowUnknownFields()               #F
    return decoder.Decode(v)                    #G
}
```

Since the body of an incoming request is a `Reader`, `DecodeJSON` takes a `Reader` to read JSON from the request body. On the other hand, `JSON` works with a `ResponseWriter` to transmit the encoded data. With the helpers ready, let's update the shortening handler.

10.3.2 Making handlers speak JSON

As the next listing shows, `Shorten` receives a request, decodes the JSON data from the request's `Body` (a `Reader`), and stores the result in the variable `link`. Once done, using `JSON`, it encodes the link's key from a map to JSON and sends it to the client. We pass a map here to send structured JSON data to the client (i.e., {"key": "something"}).

Listing 10.17 Decoding and encoding JSON (link/server.go)

```
func Shorten(links *Store) httpio.Handler {
    return func(w http.ResponseWriter, r *http.Request) httpio.Ha
        var link Link                           #A

        if err := httpio.DecodeJSON(r.Body, &link); err != nil {
            return httpio.Error(invalidRequest(err))
        }
        if err := links.Create(r.Context(), link); err != nil {
            return httpio.Error(err)
        }
        return httpio.Code(http.StatusCreated, httpio.JSON(
            map[string]string{                        #C
                "key": link.Key,                      #C
            },                                #C
        ))
    }
}

func invalidRequest(err error) error {                    #D
    return fmt.Errorf("%w: %v", bite.ErrInvalidRequest, err)
}                                    #D
```

With these changes, `Shorten` can decode and encode JSON data. For instance, say the handler receives the following JSON data:

```
{"key": "go", "url": "https://go.dev"}
```

Since the `Link` type's fields are *exported*, the handler can decode this data into a `Link`:

```
type Link struct {
    Key string        #A
    URL string        #B
}
```

Otherwise, the `json` package wouldn't decode JSON into a `Link`. For this example case above, `Link`'s `Key` field will hold "go," and the `URL` field will be "https://go.dev."

Note

The `json` package only considers exported fields while encoding and decoding.

Now that the handler can accept and send JSON data, let's give it a try. We'll post JSON data to the handler and receive a JSON response back.

```
$ curl -i localhost:8080/shorten -d '{"key": "go", "url": "https:

HTTP/1.1 201 Created
{"key":"go"}                                #B
$ curl -i localhost:8080/shorten -d '{"something": "else"}'
HTTP/1.1 400 Bad Request
invalid request: json: unknown field "something"                #
```

10.3.3 Safeguarding against DoS attacks

Let's reconsider Listing 10.17. The handler reads from the request's body (`r.Body`) without restrictions.

We can use `MaxBytesReader` to limit the number of bytes read from the request body. This might prevent clients from sending large payloads that could potentially lead to denial-of-service attacks (DoS) or overwhelm the server.

As shown below, `MaxBytesReader` is a wrapper function that wraps a `ReadCloser` and returns another, limiting the number of bytes that can be read from the wrapped one.

```
func MaxBytesReader(ResponseWriter, io.ReadCloser, int64) io.Read
```

Note

`ReadCloser` is similar to a `Reader` but also has a `Close` method. But we don't need to call `Close` as `http.Server` automatically closes the request's body when a handler returns.

As the next listing shows, `MaxBytesReader` wraps the request's `Body`. It'll stop after reading 4 KB and write an error if this limit is reached. `MaxBytesReader` has a tight integration with `http.Server` and can also tell the server to stop reading the request's body.

Listing 10.18 Using MaxBytesReader (link/server.go)

```
func Shorten(links *Store) httpio.Handler {
    return func(w http.ResponseWriter, r *http.Request) httpio.Ha
        ...

        max := http.MaxBytesReader(w, r.Body, 4_096)        #A
        if err := httpio.DecodeJSON(max, &link); err != nil {
            ...
        }

        ...
    }
```

```
}
```

Using `MaxBytesReader`, our server gains an extra layer of security to make it more robust and secure, shielding it from malicious clients that might send large amounts of data. For instance, we can easily test that it works using curl (you can use another tool of your liking).

```
$ curl localhost:8080/shorten -d "{\"key\":\"$(printf 'a%.0s' {1.

HTTP/1.1 400 Bad Request
invalid request: http: request body too large
$ curl localhost:8080/shorten -d "{\"key\":\"$(printf 'a%.0s' {1.
HTTP/1.1 400 Bad Request
invalid request: key too long (max 16)                    #B
```

10.3.4 Wrap up

- `json.Decoder` decodes JSON from an `io.Reader` into a value.
- `json.Encoder` encodes and writes a value to an `io.Writer` as JSON.
- `http.MaxBytesReader` limits reading from the request's body.

10.4 Exercises

1. Update `Health` to return JSON using `httpio` helpers.
2. Create a "/link" handler to display a link’s key, URL, creation time, and hits in JSON.
3. Adapt `httpio.Error` to output errors dynamically in text or JSON, matching the request's Content-Type (use `Request.Header().Get("Content-Type")`).
4. Implement rate-limiting middleware with a "-rate" server flag to control access frequency by IP in a specific time period.
5. Create middleware to get an API key from the request's "X-API-Key" header. Pass the API key to handlers using `Context`.
6. Create middleware restricting link shortening to requests with a valid API key. You can store the API keys in the database or in an in-memory map.
7. Implement a rate-limiting middleware using `time.Ticker`. See Chapter 7's throttler stage for an example.

In the next chapter, we'll dive into adding database support to our project.

10.5 Summary

- Middleware provides cross-cutting concerns without changing other handler code, providing increased flexibility and maintainability.
- Embedding reduces code duplication and allows composing types. However, embedded interfaces can hide the capabilities of the underlying types.
- `ResponseController` streamlines handlers to set their individual read and write timeouts and access optional hidden interfaces.
- `Context.WithValue` attaches a value to a `Context` derived from an existing one.
- `Context.Value` extracts a value from a `Context` using a key.
- `Context` keys should be unique types to prevent key collisions.
- `slog` loggers handle log records with a type that implements `slog.Handler`.
- Like `WithGroup`, `WithAttr` derives a new `Handler` from an existing one.
- The handler chaining pattern reduces repetitive code and erroneous usage. It naturally integrates with the `net/http` package without getting in the way.
- The `json` package provides JSON encoding and decoding.
- `MaxBytesReader` makes DoS attacks less likely.
- Handlers can be tested by recording their responses using `ResponseRecorder`.

11 Database and Integration Testing

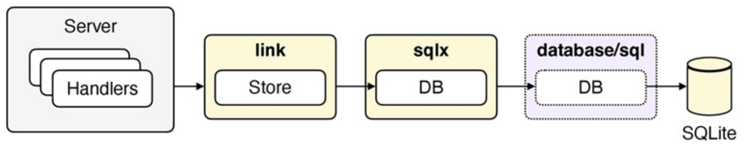
This chapter covers

- Discovering and extending the sql package.
- Structuring code with external dependencies for improved maintainability.
- Testing approaches for a reliable and stable codebase.
- Designing idiomatic interfaces to achieve modularity and composability.

In the previous chapters, our engineering team at Bite built a link server. However, there are flaws: The server stores links in memory. The product owner, Lina, wants the links preserved in a database long-term, even after the link server is restarted.

Figure 11.1 shows that the team picked the SQLite database for its practicality and lightweightness. Instead of temporarily storing links in memory, `Store` stores them permanently in SQLite using the `sqlx` and `sql` packages.

Figure 11.1 The link server uses Store to store links permanently. Store uses sqlx, which uses Go's database/sql to interact with SQLite.



Go's `sql` package can communicate with any SQL (or SQL-like, row-oriented) database. It enables our programs to work independently of the underlying SQL database technology. We'll also create the helper `sqlx` package to work with the `sql` package (which can help us augment `sql` with additional functionality, such as logging, tracing, etc.).

This upgrade will enable the server to persist links long-term. Moreover, our team will gain valuable experience working with SQL databases. Throughout the chapter, we'll dive into designing, structuring, and testing a maintainable server with database support.

Lastly, we will explore designing idiomatic and effective interfaces, which require a slightly different mindset and perspective than other object-oriented languages.

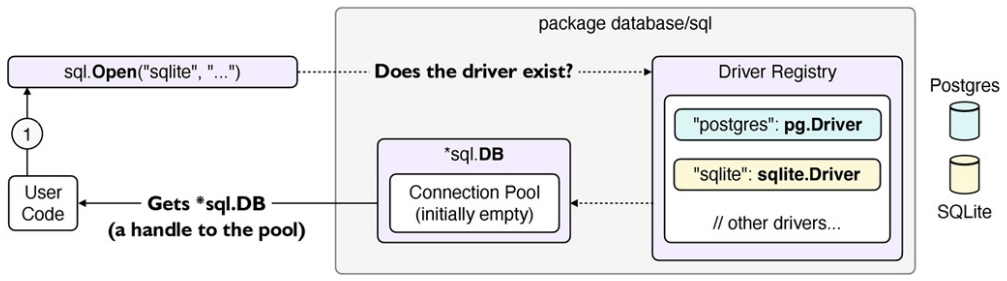
Note

Find the source code for this chapter at the following link: https://github.com/inancgumus/gobyexample/tree/main/bite.

11.1 The database/sql package

Let's start with the `sql` package. As Figure 11.2 shows, the `sql` package provides a standard interface to interact with any SQL database as long as we have the necessary *driver package*. Drivers are often generously provided by community and database vendors. We can plug the driver into the `sql` package to work with the desired database technology.

Figure 11.2 The `sql` package abstracts underlying databases with drivers. The registry stores drivers specific to a database. The pool initially has no connections.



After downloading a driver, and registering it in the `sql` package, we can use the following `Open` function to get a *handle* (`*sql.DB`) and interact with a database via a connection pool.

```
func Open(driverName string, dataSourceName string) (*sql.DB, err

// sql.Open("sqlite", "file:bite.db")
// sql.Open("mysql", "user:pass@tcp(localhost:3306)/bite")
```

Note

`sql.DB` is not a connection or database. Rather, it's an abstraction to seamlessly manage multiple database connections and perform auto-retries and error handling.

The `Open` function queries the global driver *registry* and returns a handle if the requested driver is registered. Then, we can use the handle to interact with the database. The first argument is the *driver's name*—usually, it's the same as the driver package's name to avoid collisions and confusion. The second one is the *driver-specific connection string*.

Because the `sql` package is an abstraction and only provides a standard way to communicate with SQL databases, Go does not bundle an SQL driver. `Open` fails if we don't have the necessary driver. Since our team desires to use SQLite, next, let's download and register an SQLite driver so we can obtain a database handle to work with SQLite.

Note

One can implement a driver satisfying the `sql.Driver` interface. Similar to how the `sql` package abstracts the underlying databases using drivers, Go CDK uses a similar approach to abstract cloud providers: gocloud.dev/concepts/structure. This document includes great insights into structuring code for portability using the driver pattern.

11.1.1 Downloading and registering a driver

The following command downloads the `sqlite` driver package with version 1.29.2 and adds it to our Go module. At the time of writing this chapter, this was the latest version number.

```
$ go get modernc.org/sqlite@v1.29.2
```

Tip

For a list of all SQL drivers, visit https://go.dev/wiki/SQLDrivers.

Now that the driver is downloaded, we can connect to SQLite by registering it in the `sql` package. To do that, we must import the driver, as shown in the next listing.

Listing 11.1 Registering the driver (sqlx/sqlx.go)

```
package sqlx

import (
    _ "modernc.org/sqlite"    #A
)
```

We've imported the driver with a *blank identifier* (_) to tell the compiler that we won't use the driver package's name. This importing style is known as a *side-effect or blank import*.

Since we won't use the driver directly in our code but rather the `sql.DB` type to interact with the database, this step is necessary. If we skip it, the compiler panics with an "*imported and not used*" error, or our text editor removes the import because we're not using the driver package's functions or types in the current file. Next, we'll connect to SQLite.

Tip

Run `go mod tidy` to clean up the Go module dependency graph after saving the sqlx.go file. Read more about the tidy command here: https://go.dev/ref/mod#go-mod-tidy.

Dive deep: Blank imports and init()

Let's discuss how blank imports and the magical `init` function work a bit more.

```
package foo
func init() {
    // do something global and have side effects!
}
```

When we import a package, Go automatically executes any `init` functions declared in the package (there can be more than one, and they run in the order they appear). SQL drivers often contain at least one `init`, enabling them to register with the global SQL driver registry.

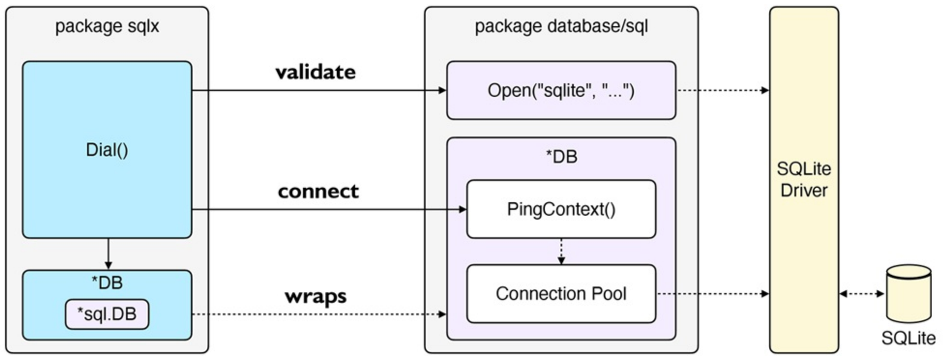
Ideally, we should avoid `init` functions as they're automatically and implicitly called. They can lead to code that is challenging to wrap our head around and maintain. The `sql` package uses `init` functions to maintain Go 1 backward compatibility and is stuck with them. Refer to the Go specification for more information on `init`: go.dev/ref/spec#Package_initialization.

11.1.2 Opening up a connection pool

Now that we're introduced to `sql` and got the driver, it's time to connect to SQLite. We'll use the `sqlx` package to establish a connection to the database. The `sqlx` package simplifies how our program interacts with the database and lets us optimize performance and add application-specific functionality when needed such as, tracing database operations.

Figure 11.3 shows how to connect to SQLite using the `sql` and `sqlx` packages.

Figure 11.3 sqlx.Dial opens a connection pool using sql.Open, pings the database using PingContext, and wraps and returns *sql.DB as *sqlx.DB.



The `Dial` function calls `sql.Open` with the driver's name. `Open` doesn't establish a connection and only validates if the driver exists in the registry. Then, it returns a handle (`*sql.DB`) to the connection pool. This pool, initially, has no established connections. Once `Dial` gets the pool handle, it calls `PingContext` to establish a connection to the database (for real). Lastly, `Dial` wraps and returns `*sql.DB` in a new `*sqlx.DB`, which will act exactly like `*sql.DB`.

In summary, `Dial` returns a new `*sqlx.DB` that wraps a connection pool handle (`*sql.DB`). The connection pool had no connections at the beginning. We obtain the first connection and validate if the database is reachable using the pool's `PingContext` method.

Tip

We should maintain `*sql.DB` we obtained from `Open` throughout our program's lifespan since it includes a connection pool to the database. We rarely (almost never) close it since the `sql` package handles connection management automatically.

11.1.3 Dialing the database

Since we learned the basics, let's look at the next listing. We declare the new `DB` type and the `Dial` function to open a new connection pool and ping the database. The function takes a driver name along with the DSN (data source name)—this means our team can support multiple database drivers in the future if they ever need to use a different SQL database.

Listing 11.2 Dialing the database (sqlx/sqlx.go)

```
package sqlx

import (
    "context"
    "database/sql"
    "fmt"

    _ "modernc.org/sqlite"
)

const DefaultDriver = "sqlite"

type DB struct {
    *sql.DB                             #A
}

func Dial(ctx context.Context, driver, dsn string) (*DB, error) {
    db, err := sql.Open(driver, dsn)
    if err != nil {
        return nil, fmt.Errorf(
            "opening database driver %q by %q: %w",
            driver, dsn, err,
        )
    }
    if err := db.PingContext(ctx); err != nil {
        return nil, fmt.Errorf("pinging database: %w", err)
    }

    return &DB{DB: db}, nil
}
```

The `sqlx` package serves as a bridge between our code and SQL databases.

The `sqlx.DB` type embeds `sql.DB`, allowing it to effectively behave like the actual `sql.DB` type. This means all `sql.DB` methods will be available through `sqlx.DB`. We'll soon see how this approach streamlines code while modifying the `Store` type.

The `Dial` function establishes a connection to SQLite using the driver. This way driver import happens in a single package. This enables us to take control of the driver version changes and prevent collisions in places that might import an identical driver with the same name.

Errors should tell a story

`Dial` errors tell a story: "*opening*" and "*pinging*". The usage of the present continuous tense is not accidental and allows us to figure out why and how errors occurred.

For example, consider the following error message:

```
"opening database driver "sqlite" by "...": sql: database is clos
```

This error message tells a better story, and is less noisy and rich than the following:

```
"failed to open database: could not dial the database: cannot pin
```

It's already clear there is a failure because it's an error!

So, we should avoid unnecessarily using "*cannot*", "*failed*", "*error*", etc. in error messages. This way, we can reduce noise and improve debugging. We can, of course, use one of these words once at the end of an error chain (the last error that doesn't wrap another error).

Trivia: See my old tweet about my stance on this: https://tinyurl.com/errors-as-stories

11.1.4 Testing the connection

We added `sqlx.DB` that wraps `sql.DB` for consistent access to SQL databases.

Now that we have the necessary foundation and haven't yet integrated `sqlx` into the link server, writing a test seems like the best way to verify the `Dial` function. The next listing shows a test for `Dial`. The DSN `":memory:"` is special for SQLite and runs a new in-memory database (non-persistent, which is a nice option especially for testing).

Listing 11.3 Testing Dial (sqlx/sqlx_test.go)

```
package sqlx

import "testing"

func TestDial(t *testing.T) {
    db, err := Dial(
        context.Background(),
        DefaultDriver,
        ":memory:",
    )
    if err != nil {
         t.Errorf("got err %q, want nil", err)    #A
    }
    if db == nil {                #A
         t.Error("got nil, want non-nil")
    }
}
```

Once we run it, we should see the test passes. As an exercise, you can try it with another SQL driver name and see how it fails. Let's move to applying the database schema next.

```
$ go test ./sqlx -v
--- PASS: TestDial
```

We can now apply the database schema.

Exercise

Add a cancel context and cancel it before calling `Dial` to see what happens.

Exercise

Try using another SQL driver name and see how it fails.

11.1.5 Applying the schema

Next up for our team is to define a database schema for saving links. As shown in the next listing, the schema defines the `links` table with a short key and original URL columns. For convenience, we set up the schema to create the `links` table only once if it doesn't exist.

Listing 11.4 Defining the schema (sqlx/schema.sql)

```
CREATE TABLE IF NOT EXISTS links (
    short_key VARCHAR(16) PRIMARY KEY,
    uri       TEXT NOT NULL
);
```

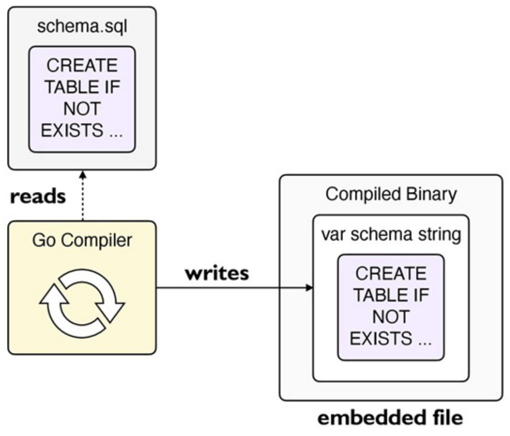
We could include this schema in a constant in the source code. However, keeping it in an SQL file is more practical, so our text editor can highlight and even lint the SQL syntax.

Since the schema is ready, we'll include it in the `sqlx` package to execute it to create a database table each time we open a connection to the database. Because we set up the schema not to recreate the `links` table each time we execute it, rerunning the schema on the database will be a safe operation and ensure that we'll always have the schema ready.

File embedding

The question is, how do we get the schema file into the `sqlx` package? That's where a nice Go feature called *embedding* comes in. It lets us include a file in a variable. See Figure 11.4.

Figure 11.4 The compiler embeds the file into the compiled binary.



The compiler allows embedding when we import the `embed` package for its side effects, so it knows we'll embed file(s). Then, we can embed the schema file using the *embed directive*:

```
import _ "embed"        #A

...
//go:embed schema.sql    #B
var schema string        #B
```

With the embedding directive, the compiler reads "`schema.sql`" while compiling and puts the file's entire content into `schema`. Since the compiler already includes the file's content within the final binary, we can deploy our program without the schema file. This streamlines developing and deploying our program. We won't need to ship a schema to users.

See the next listing where we embed schema.sql's content into the `schema` variable.

Listing 11.5 Applying the schema (sqlx/sqlx.go)

```
package sqlx

import (
    ...

    _ "embed"
)

//go:embed schema.sql
var schema string

const DefaultDriver = "sqlite"

type DB struct {
    *sql.DB
}

func Dial(ctx context.Context, driver, dsn string) (*DB, error) {
    db, err := sql.Open(driver, dsn)
    if err != nil {
        return nil, fmt.Errorf(
            "opening database driver %q by %q: %w",
            driver, dsn, err,
        )
    }
    if err := db.PingContext(ctx); err != nil {
        return nil, fmt.Errorf("pinging database: %w", err)
    }
    if _, err := db.ExecContext(ctx, schema); err != nil {    #A
        return nil, fmt.Errorf("applying schema: %w", err)
    }

    return &DB{DB: db}, nil
}
```

Once we embed the schema, we run it using `ExecContext` method:

```
func (db *DB) ExecContext(ctx context.Context, query string, args
```

`ExecContext` grabs a connection from the pool, executes the query, and then returns the connection to the pool. As we can see from its signature, we can pass an arbitrary number of arguments of any type to the `Exec` method. Since we don't need to provide parameters to the database while running the schema, there's no need to pass any parameters for now.

Note

Learn more about embedding at the link: https://pkg.go.dev/embed.

11.1.6 Wrap Up

Our team took the first steps of adding database support to their link server project using Go's `sql` package. They created a mediator package called `sqlx`, opened a database pool, pinged it for a new connection, embedded an SQL schema file, and executed it.

Let's wrap up.

- The `sql` package provides a standard API, abstracting SQL databases.
- `sql.Open` verifies a SQL driver and returns `sql.DB` with an empty connection pool.
- Go doesn't bundle SQL drivers and `Open` fails without a SQL driver.
- `sql.DB` has a connection pool and an auto-retry mechanism.
- `DB.PingContext` establishes a connection to the database.
- `DB.ExecContext` executes a statement on the database. Like `PingContext`, it might establish a connection to the database or reuse an existing one from the pool.
- The `embed` package lets us embed files directly into the program binary.

Optimizing the connection pool

Before concluding this section, let's take a look at how we can optimize the connection pool. When it comes to managing connections, `sql.DB` provides several knobs to fine-tune the connection's pool's behavior for optimal performance. There are idle and in-use connections within the pool, and the pool checks for an idle connection when executing a database task. If none are available, a new connection is created. We can tune how this happens.

To manage the maximum number of connections allowed in the pool, we can use `SetMaxOpenConns` and `SetMaxIdleConns`. It's important to note that setting these values too high can lead to performance issues and inefficiencies, as idle connections consume memory and may become unusable. There is also a limit to how much work the database server can do simultaneously. If we create many connections because each query we send takes a long time to respond, adding more queries will likely overload the server.

There are also `SetConnMaxLifetime` and `SetConnMaxIdleTime` methods that we can use to determine how long a connection can be reused and how long it can remain idle, respectively. Setting `MaxIdleConns` too high can cause unusable connections and wasted resources. Instead of maximizing the number of connections available, optimizing the connection pool settings by thoroughly testing and benchmarking them is best.

For more details on tuning, visit go.dev/doc/database/manage-connections

11.2 Storage service

Our program can now connect to SQLite and apply the schema. The next step is to persist links on SQLite. Let's remember `link.Store` from the previous chapters.

```
package link

type Store struct {
    mu    sync.RWMutex
    links map[string]Link
}

func NewStore() *Store {
    return &Store{links: make(map[string]Link)}
}

func (s *Store) Create(_ context.Context, link Link) error {
    ...
}

func (s *Store) Retrieve(_ context.Context, key string) (Link, er
    ...
}
```

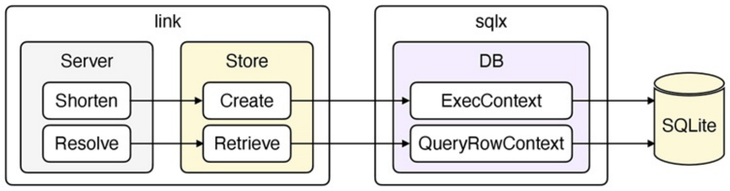
We had created this type to persist links temporarily in memory. HTTP handlers use it.

Rather than storing links in memory, in this section, we'll modify `Store` and its methods to persist links on SQLite. We'll replace the mutex and the map with `sqlx.DB`. We'll also modify `Create` and `Retrieve` to include SQL queries and execute them on SQLite.

Figure 11.5 overview our ultimate plan for `Store`:

- `Create` *inserts* a link into the database using `ExecContext`.
- `Retrieve` uses `QueryRowContext` to get a link from the database by its short key.

Figure 11.5 Handlers use Store to create and retrieve links in SQLite; internally, Store uses the sqlx.DB type to execute queries on the database.



Both methods use the `sqlx.DB` type to execute SQL queries.

Also, recall from Listing 11.2 that `sqlx.DB` embeds the `sql.DB` type, borrowing its methods. That's why we see `sql.DB`'s `ExecContext` and `QueryRowContext` on `sqlx.DB`.

`Create` and `Retrieve` let handlers speak business language rather than `ExecContext` and `QueryRowContext`. This decouples handlers from the underlying technology. Because our program clearly isolates responsibilities, we can persist links to the database by simply changing these methods. We don't need to change the rest of the program. Moreover, using the business language makes understanding what the handlers do is straightforward.

Getting ready

Before getting started, let's remove the map and add `sqlx.DB` into `Store`. See the next listing. This way, `Create` and `Retrieve` can call specific `DB` methods to interact with the database. We're also gutting `Create` and `Retrieve` to prepare for the upcoming changes.

Listing 11.6 Adding sqlx.DB into Store (link/Store.go)

```
type Store struct {
    db *sqlx.DB
}

func NewStore(db *sqlx.DB) *Store {
    return &Store{db: db}
}

func (s *Store) Create(ctx context.Context, link Link) error {
    return nil
}

func (s *Store) Retrieve(_ context.Context, key string) (Link, er
    return Link{}, nil
}
```

We're now ready to rewrite these methods from scratch.

11.2.1 Persisting links

Let's first focus on persisting links using `Store.Create`. This subsection explores writing and executing SQL queries and inserting data into the database. We'll also look at creating test helpers using `t.Helper` and execute after-test clean-up code using `t.Cleanup`.

Executing queries without returning rows

Let's update `Create` to execute a query on the database. See the next listing.

Listing 11.7 Modifying Create (link/Store.go)

```
func (s *Store) Create(ctx context.Context, link Link) error {

    if err := validateNewLink(link); err != nil {
        return fmt.Errorf("%w: %w", bite.ErrInvalidRequest, err)
    }

    const query = `
        INSERT INTO links (
            short_key, uri
        ) VALUES (
            ?, ?                            #A
        )`
    _, err := s.db.ExecContext(ctx, query, link.Key, link.URL)
    if err != nil {
        return fmt.Errorf("creating link: %w", err)
    }

    return nil
}
```

`Create` inserts the link into the database using `ExecContext`. Once the query is executed, `ExecContext` automatically returns the connection to the pool for later reuse (see the connection pool from Section 11.1.2 for more details).

The question marks in the query are called *placeholders* and let us substitute the link key and URL when running the query. We're avoiding handcrafted SQL parameters and using placeholders to make SQL injection attacks less likely.

The downside is that, although we use standard SQL, the placeholder syntax is database-specific. Luckily, MySQL uses the same syntax (so our SQL is MySQL-ready!). But it's not ready for Postgres as it uses $1, $2, etc.

Tip

The sqlx package (not ours!) extends Go's sql and can modify placeholders to work across SQL databases. It also provides other useful features like saving a query's results directly into struct values. Visit the link to learn more about it: https://github.com/jmoiron/sqlx.

Tip

The sql package supports prepared statements for efficiently rerunning the same database statements. See the link for details: https://go.dev/doc/database/prepared-statements

Writing a test helper for connecting to the database

Before testing `Create`, let's add a helper to make connecting to a test database easier.

`Dial` in the next listing lets tests get an in-memory database and automatically close it when they finish. We're adding `Dial` to `sqlxtest` as we'll only import this package from tests.

Listing 11.8 Adding a test dialer helper (sqlx/sqlxtest/sqlxtest.go)

```
package sqlxtest


import (
    "context"
    "testing"

    "github.com/inancgumus/gobyexample/bite/sqlx"
)

func Dial(ctx context.Context, tb testing.TB) *sqlx.DB {
    tb.Helper()                         #A

    dsn := fmt.Sprintf(
        "file:%s?mode=memory&cache=shared",
        tb.Name(),                     #B
    )

    db, err := sqlx.Dial(ctx, sqlx.DefaultDriver, dsn)
    if err != nil {
        tb.Fatalf("dialing test db: %v", err)
    }
    tb.Cleanup(func() {                #C
        if err := db.Close(); err != nil {        #C
            tb.Log("closing test db:", err)    #C
        }                     #C
    })                         #C

    return db
}
```

We give each test a unique in-memory database (mode=memory) that is shared (cache=shared) by all database connections that happen in the same test. `tb.Name` returns the current test name that calls `Dial` (e.g., it returns "TestFoo" if `TestFoo` calls `Dial`).

This way, we can run tests concurrently. Each database operation in the same test will use the same database as long as the test calls `Dial` once. We could have set the database name with a random identifier, but using the test name is good enough for now.

Calling the `Helper` method marks the `Dial` function as a *test helper*. When printing file and line information from the caller test (e.g., `TestCreate`), `Dial`'s file and line information will be skipped. `Dial` will act as if it were part of the same test that calls it. We typically want to see the original test code that fails (or logs) rather than a line in a helper function.

For instance, say `TestStoreCreate` calls `Dial`, and `Dial` fails:

```
--- FAIL: TestStoreCreate (0.00s)
    Store_test.go:22: dialing test db: pinging database: ...: out
```

And the 22nd line in Store_test.go looks like this:

```
func TestStoreCreate(t *testing.T) {
    ...
    store := NewStore(sqlxtest.Dial(ctx, t)) // <---
    ...
}
```

The test log reports that `TestStoreCreated` failed while dialing the database. Instead of seeing the fatal error originating from the `Dial` function, the test that calls `Dial` fails. The reported file name and line information is about the caller test, not `Dial`.

Calling the `Cleanup` method is useful for doing after-test clean-up work. It's similar to the `defer` statement. However, there is a crucial difference. `Cleanup` is called once the test that calls `Dial` finishes rather than the `Dial` *function returns*. Otherwise, the database connection would be closed before the caller test finishes, which wouldn't be what we want.

With the new test helper, connecting to a test database becomes straightforward. The `Dial` helper connects to a unique in-memory database with the caller test's name for convenience.

Tip

Using `testing.TB` instead of `*testing.T` lets us use the `Dial` function both in tests and benchmarks. `testing.TB` is an interface that both `*testing.T` and `*testing.B` types satisfy.

Testing

See the next listing that shows how to test the `Create` method with the `Dial` test helper.

We first set up the necessary variables and then test the `Store`'s `Create` method.

Listing 11.9 Testing Create (link/Store_test.go)

```
package link

import (
    "context"
    "testing"

    "github.com/inancgumus/gobyexample/bite/sqlx/sqlxtest"
)

func TestStoreCreate(t *testing.T) {
    t.Parallel()

    ctx := context.Background()
    link := Link{Key: "go", URL: "https://go.dev"}

    t.Run("ok", func(t *testing.T) {
        t.Parallel()

        store := NewStore(sqlxtest.Dial(ctx, t))    #A
        if err := store.Create(ctx, link); err != nil {
            t.Errorf("Create(%q) err = %v, want nil", link.Key, e
        }
    })
}
```

We've written a test to see if we can create a link. Let's see if it works.

```
$ go test ./link -run=TestStoreCreate -v
--- PASS: TestStore/ok
```

We'll soon add more tests to test `Create` and `Retrieve`.

Testing for duplicate links

Let's verify what happens when we add a link with a duplicate key. See the next listing.

Listing 11.10 Testing if the link exists (link/Store_test.go)

```
func TestStoreCreate(t *testing.T) {
    ...

    t.Run("err_exists", func(t *testing.T) {
        t.Parallel()

        store := NewStore(sqlxtest.Dial(ctx, t))
        if err := store.Create(ctx, link); err != nil {
            t.Errorf("Create(%q) err = %v, want nil", link.Key, e
        }
        if err := store.Create(ctx, link); !errors.Is(err, bite.E
            t.Errorf("Create(%q) err = %v, want %v", link.Key, er
        }
    })
}
```

Let's run the test to see if `Create` returns `ErrExists` when adding a duplicate link.

```
$ go test ./link -run=TestStoreCreate/err_exists -v
Create("go") err = "creating link: constraint failed: UNIQUE cons
```

The test failed because `Create` isn't immune to duplicate links. At least we see an error. Since the schema's `short_key` column is a primary key, like most databases, SQLite is designed to handle such an issue and reports an error through the driver.

Extracting errors with errors.As

The previous error was clearly a unique constraint error. However, in code, it can be difficult to determine whether an error is a constraint error. Parsing error message strings can also be cumbersome as they can change when drivers update. Instead, we should detect the actual error types. The good news is that type assertion comes to the rescue! We can extract the driver error from the error value returned by `Create`.

For instance:

```
serr, ok := err.(*sqlite.Error)
```

However, doing so can't find the SQLite error if it's buried in the error chain somewhere. An idiomatic approach is to use `errors.As` to extract the error from the chain. `As` extracts and assigns the target error to a variable and returns `true` if the error is in the error chain:

```
var serr *sqlite.Error
ok := errors.As(err, &serr)    #A
```

In the next listing, we're adding the unique constraint error-checking logic as a function, as other future services in our company might need a similar check. `IsPrimaryKeyViolation` uses `As` to extract the driver error and save it into a variable. Then, we query the database error code and return `true` if it's a primary key constraint error code. Since `sqlite.Error` is specific to the SQLite driver, we're coupling our code to the driver.

Listing 11.11 Adding a constraint detector (sqlx/sqlx.go)

```
package sqlx

import (
    // ...other imports here...

    "modernc.org/sqlite"                              #A
    sqlite3 "modernc.org/sqlite/lib"
)

...

func IsPrimaryKeyViolation(err error) bool {
    var serr *sqlite.Error                           #B
    if errors.As(err, &serr) {                           #C
        return serr.Code() == sqlite3.SQLITE_CONSTRAINT_PRIMARYKE
    }
    return false
}
```

By checking for driver-specific errors, our `sqlx` package is now tightly bound to the driver package. An alternative approach would be to use transactions to query the database before inserting a row (see https://go.dev/doc/database/execute-transactions). However, this would involve more complex code and require additional round trips to the database. Also, it wouldn't fix the problem. Two concurrent server requests might have inserted a row between the first query checking for existence and the second failing to insert.

The current approach strikes a nice balance. Since SQLite is already coupled to the driver, it makes sense to leverage that relationship. The `IsPrimaryKeyViolation` function is especially helpful because it contains all the driver-specific code in one place. Instead of trying to decipher cryptic SQLite errors, we can use `IsPrimaryKeyViolation` to detect duplicate links effectively. It's a much simpler and cleaner solution.

Moreover, we could detect the SQL driver type and then check for the specific database error code specific to that driver. This would be straightforward since we confine the error detection to a single function instead of dispersing the logic throughout our program.

In the following listing, we use `IsPrimaryKeyViolation` in the `Create` method.

Listing 11.12 Using the constraint detector (link/Store.go)

```
var ErrLinkExists = fmt.Errorf("link %w", bite.ErrExists)

func (s *Store) Create(ctx context.Context, link Link) error {
    ...

    _, err := s.DB.ExecContext(ctx, query, link.Key, link.URL)
    if sqlx.IsPrimaryKeyViolation(err) {
        return ErrLinkExists
    }
    if err != nil {
        return fmt.Errorf("creating link: %w", err)
    }
    return nil
}
```

Let's give the test another go and see if the `Create` method is capable of detecting dups.

```
$ go test ./link -run=TestStore/create/exists
PASS
```

11.2.2 Retrieving links

Having looked at inserting a link into the database, let's now dive into `Retrieve` to fetch a link from the database. The next listing retrieves the link by the short key.

Listing 11.13 Adding the Retrieve method (link/Store.go)

```
var (

    ErrLinkExists   = fmt.Errorf("link %w", bite.ErrExists)
    ErrLinkNotExist = fmt.Errorf("link %w", bite.ErrNotExist)
)

func (s *Store) Retrieve(ctx context.Context, key string) (Link,
    if err := validateLinkKey(key); err != nil {
        return Link{}, fmt.Errorf("%w: %w", bite.ErrInvalidReques
    }

    const query = `
        SELECT uri
        FROM links
        WHERE short_key = ?`

    var url string

    err := s.db.QueryRowContext(ctx, query, key).Scan(&url)
    if errors.Is(err, sql.ErrNoRows) {                #B
        return Link{}, ErrLinkNotExist
    }
    if err != nil {
        err := fmt.Errorf("retrieving link by key %q: %w", key, e
        return Link{}, err
    }
    return Link{
        Key: key,
        URL: url,
    }, nil
}
```

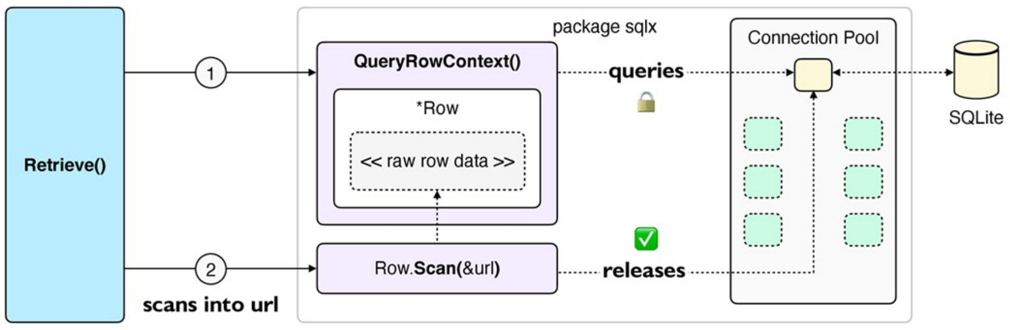
When we want to pull one row from the database, `QueryRowContext` is the way to go.

The `Retrieve` function validates the key, then calls `QueryRowContext` to send the SQL query to the database. After that, it passes the `url` variable's pointer to the `Scan` method. `Scan` then parses and converts the query result to a `string` and updates the `url` variable.

We're also translating the `sql` package's `ErrNoRows` error to our domain error (`ErrNotExist`) for consistency with the rest of our program.

Figure 11.6 illustrates how the `Retrieve` function works. `QueryRowContext` queries the database and reserves the connection until `Scan` is called. Once we call `Scan`, it injects the link URL into the `url` variable and returns the connection to the pool.

Figure 11.6 Retrieve calls QueryRowContext to retrieve the URL from the database. The connection is reserved until Scan is called. Calling Scan copies the previously loaded raw row data into the url variable and returns the connection to the pool.



Keep in mind that, we risk leaking database connections if we don't call `Scan` after `QueryRowContext`. `Scan` both gets the data and releases the connection. This boosts performance and reduces resource usage by reusing the connection for other queries.

Note

If the SQL query returns multiple rows, `QueryRowContext` loads all of them from the database into memory, and fortunately, `Scan` returns the first row and discards the rest.

Note

For fetching multiple rows, we can use `QueryContext`. Remember to release the returned `Rows`. Otherwise, the database connection will leak.

Testing

The next listing shows a test to verify `Retrieve`. The first subtest retrieves the link and expects no error. The second retrieves a non-existing link and expects an error.

Listing 11.14 Testing the Retrieve method (link/Store_test.go)

```
func TestStoreRetrieve(t *testing.T) {

    t.Parallel()

    ctx := context.Background()

    link := Link{Key: "go", URL: "https://go.dev"}

    t.Run("ok", func(t *testing.T) {
        t.Parallel()

        store := NewStore(sqlxtest.Dial(ctx, t))
        if err := store.Create(ctx, link); err != nil {
            t.Errorf("Create(%q) err = %v, want nil", link.Key, e
        }
        got, err := store.Retrieve(ctx, link.Key)
        if err != nil {
            t.Errorf("Retrieve(%q) err = %v, want nil", link.Key,
        }
        if got != link {
            t.Errorf("Retrieve(%q) = %#v, want %#v", link.Key, go
        }
    })

    t.Run("err_not_exist", func(t *testing.T) {
        t.Parallel()
        store := NewStore(sqlxtest.Dial(ctx, t))
        _, err := store.Retrieve(ctx, "void")
        if !errors.Is(err, bite.ErrNotExist) {
            t.Errorf(`Retrieve(void) err = %v, want %v`, err, bit
        }
    })
}
```

We've written a subtest to see `Retrieve` can fetch a link and another one also to check if `Retrieve` returns `ErrNotExist` for a non-existent link. Let's run the test to see if they work.

```
$ go test ./link -run=TestStoreRetrieve -v
--- PASS: TestStoreRetrieve/ok
--- PASS: TestStoreRetrieve/err_not_exist
```

11.2.3 Wrap up

Our team has modified `Store` to persist links on SQLite. `Store` wraps `sqlx.DB` and offers a high-level storage service for the program. `Create`, using `ExecContext`, adds a `Link` to the database. `Retrieve` uses `QueryRowContext` and `Scan` to grab a `Link` from the database. Both methods are protected against SQL injection attacks using placeholders.

Let's wrap up.

- `ExecContext` runs a query and doesn't return database rows.
- `QueryRowContext` runs a query and returns database row(s). Calling `Scan` scans the loaded rows, extracts and automatically transfers the data to a variable.
- Using placeholders in SQL queries makes SQL injections attacks less likely.
- Calling `*testing.T`'s `Helper` marks the caller function as a test helper.
- Calling `*testing.T`'s `Cleanup` is useful for doing after-test clean-up work.
- Both `*testing.T` and `*testing.B` satisfy the `testing.TB` interface.
- Using a type assertion to extract errors from an error chain is ineffective.
- `errors.As` looks up an error in the error chain and sets it to a variable if matches.

11.3 Extending the sql package

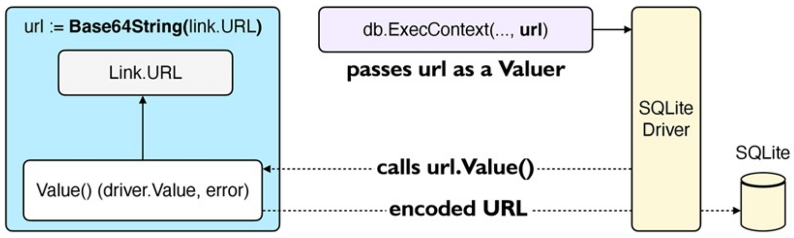
In the previous section, our team implemented `link.Store`'s `Create` and `Retrieve` methods to save and retrieve links from the database. However, just when they thought they were done, product owner Lina had yet another request: she wanted URLs to be saved in the database with *Base64 encoding* to prevent special character issues in URL strings.

Note

This is an imaginary request to showcase the `Valuer` and `Scanner` interfaces. Saving into the database without encoding might be more effective in a real-world program.

To satisfy Lina's request, our team will implement the `driver.Valuer` interface to modify the `url` column before sending it to the database. Check out Figure 11.7.

Figure 11.7 Using Valuer to encode a value before saving it to the database.



```
type Valuer interface {

    // Value returns a driver Value.
    Value() (Value, error)
}
// ...
type Base64String string

func (s Base64String) Value() (driver.Value, error) { ... }
// ...
_, err := s.db.ExecContext(ctx, query, link.Key, sqlx.Base64Strin
```

Since we want to encode `URL` before saving it, we're using a type called `Base64String` to satisfy `Valuer`. Then, we're converting and passing `URL` as

`Base64String` to `ExecContext`. The driver automatically calls the `Value` method to save the encoded URL to the database.

Additionally, the `sql.Scanner` interface works in the opposite direction. `Base64String` will satisfy `Scanner` to decode the URL after retrieving (e.g. using `Scan`) it from the database.

```
type Scanner interface {

    // Scan assigns a value from a database driver.
    Scan(src any) error
}
```

To sum up, `Valuer` enables us to modify a value before it is written to the database, while `Scanner` does the opposite. We can use `Valuer` to encode URLs in Base64 before saving and `Scanner` to decode them after retrieving them from the database.

11.3.1 Satisfying Valuer and Scanner

As the next listing shows, we declare the `Base64String` type to satisfy `Valuer` and `Scanner`.

- `Value` satisfies `Valuer` and encodes to Base64 from a string.
- `Scan` satisfies `Scanner` and decodes from Base64 to string.

We use Go's `base64` package to encode and decode. `StdEncoding` encodes using Base64 encoding as defined in RFC 4648.

- `EncodeToString` encodes a byte slice to a string.
- For decoding, we use `StdEncoding`'s `DecodeString` method.

Since a database column can be of different types, `Scan` expects any type. Using a type assertion (`src.(string)`), we're ensuring what we receive from the driver is a string.

Listing 11.15 Satisfying Valuer, Scanner, and Stringer (sqlx/base64.go)

```
package sqlx
```

```
import (
    "database/sql/driver"
    "encoding/base64"
    "fmt"
)

type Base64String string

func (s Base64String) Value() (driver.Value, error) {    #A
    dst := []byte(s)
    return base64.StdEncoding.EncodeToString(dst), nil    #A
}

func (s *Base64String) Scan(src any) error {        #B
    ss, ok := src.(string)               #C
    if !ok {
        return fmt.Errorf("%q is %T, not string", ss, src)
    }
    dst, err := base64.StdEncoding.DecodeString(ss)        #D
    if err != nil {
        return fmt.Errorf("decoding %q: %w", ss, err)
    }

    *s = Base64String(dst)               #E

    return nil
}

func (s Base64String) String() string {
    return string(s)
}
```

While `Value` encodes a string to a Base64 string, `Scan` decodes a Base64 encoded string. `Base64String` is a `Scanner`, `Valuer`, and `Stringer` because it satisfies all of these interfaces. This is made possible by Go's *implicit interfaces*, which allow a type to fulfill an interface's methods without explicitly stating that it implements the interface.

11.3.2 Encoding and decoding

We can now integrate `Base64String` into `Store`'s `Create` and `Retrieve` methods. In the next listing, we convert the URL field to `Base64String` to encode and decode URLs.

Listing 11.16 Using Base64String (link/Store.go)

```
func (s *Store) Create(ctx context.Context, link Link) error {
    // ...

    url := sqlx.Base64String(link.URL)                #A
    _, err := s.DB.ExecContext(ctx, query, link.Key, url)

    // ...
}

func (s *Store) Retrieve(ctx context.Context, key string) (Link,
    // ...

    var url sqlx.Base64String                         #C
    err := s.db.QueryRowContext(ctx, query, key).Scan(&url)

    // ...

    return Link{
        Key: key,
        URL: url.String(),                            #D
    }, nil
}
```

`Create` passes the URL as `Base64String` to `ExecContext` to automatically encode the URL before saving the URL into the database. Similarly, `Retrieve` passes `url` as `Base64String` to `Scan` to decode the Base64 encoded database column (the url column) to a URL.

We can now create and retrieve URLs in Base64 encoded and decoded formats. Let's run the tests and see if everything's running smoothly.

```
$ go test ./link -run=TestStore -v
--- PASS: TestStoreCreate (0.00s)
    --- PASS: TestStoreCreate/err_exists (0.00s)
    --- PASS: TestStoreCreate/ok (0.00s)
--- PASS: TestStoreRetrieve (0.00s)
    --- PASS: TestStoreRetrieve/ok (0.00s)
    --- PASS: TestStoreRetrieve/err_not_exist (0.00s)
```

We don't have to modify the tests, as the encoding and decoding are automatic. We can use this technique to encode and decode any column from the database.

11.3.3 Wrap up

- `driver.Valuer` allows processing a value before delivering it to the database.
- `sql.Scanner` allows processing a value after getting it from the database.
- The `base64` package provides Base64 encoding and decoding.
- `EncodeToString` encodes a byte slice to Base64.
- `DecodeString` decodes a Base64 encoded string.

11.4 Updating the daemon

Since `Store` is ready, let's integrate it into `linkd`. Recall from the previous chapters that `linkd` launches the link HTTP server. See the next listing.

Listing 11.17 Updating the daemon (cmd/linkd/linkd.go)

```
package main

...

func main() {
    const addr    = "localhost:8080"
    const timeout = 10 * time.Second
    const dbDSN   = "file:bite.db?mode=rwc"            #A

    ...

    ctx := context.Background()                     #B
    db, err := sqlx.Dial(ctx, sqlx.DefaultDriver, dbDSN)       #
    if err != nil {                             #B
        log.Error("connecting to database", "message", err)
        return                                  #B
    }                                     #B

    links := link.NewServer(link.NewStore(db))        #C

    ...
}
```

We connect to SQLite using `sqlx.Dial`. SQLite will store the shortened links in a file (bite.db). The "rwc" mode means Read-Write-Create: creates the database file if it doesn't exist and opens it for reading and writing. Lastly, we pass the `sqlx.DB` to the server.

With minimal changes, we've upgraded the link server to SQLite. Let's try it. Remember to run the server. Then, send the requests to the server from another session.

```
$ go run ./cmd/linkd

level=INFO msg="starting the server" app=linkd addr=localhost:808

$ curl -i localhost:8080/shorten -d '{"key": "go", "url": "https:
HTTP/1.1 201 Created
{"key":"go"}

$ curl -i localhost:8080/shorten -d '{"key": "go", "url": "https:
HTTP/1.1 409 Conflict
link already exists

$ curl -i localhost:8080/r/go
HTTP/1.1 302 Found
Location: https://go.dev.
```

The link server is ready to meet with the real world. The project is complete.

11.5 Testing

Now that the project is complete, we'll consider some options for testing the server. Different approaches might help us decide which to use for the case.

We'll look at the following approaches:

- Testing against the real thing.
- Faking with interfaces.

Before getting into these approaches, let's discuss when to use unit and integration tests.

Isolating parts of a program with an interface

Programmers often debate whether testing HTTP handlers should involve a direct database connection or not. One solution is to implement interfaces to abstract service logic from handlers. For instance, using an interface instead of `Store` in handlers:

```
package link

type CreateRetriever interface {
    Create(ctx context.Context, link Link) error        #A
    Retrieve(ctx context.Context, key string) (Link, error)    #A
}

func NewServer(links CreateRetriever) *Server { .. }    #B
```

This separation allows injecting any type with a `Create` and `Retrieve` method (e.g., `Store`). It also makes it possible to pass a test-double (to use a type in tests in place of the real type, such as fakes, stubs, mocks, etc.) to the handlers to test them in isolation.

Should we separate `Store` from handlers *only* for testing purposes?

Unit vs. integration testing

Unit testing code with pure logic is effective because we can test logic that doesn't depend on external components in *isolation*. That's why we isolated the business logic (i.e., `validateNewLink`) into separate functions to ease testing edge cases.

```
package link
func validateNewLink(link Link) error {
    if err := validateLinkKey(link.Key); err != nil {
        return err
    }
    // ...
    return nil
}
```

Note

Isolate business logic as much as possible into pure functions to unit test them.

Unlike pure functions, unit testing higher-level components like `Store` and handlers aren't practical since they don't solely contain isolated and standalone logic: handlers rely on `Store` for persisting links, which requires a database. They *assemble* (integrate) internal and external parts of our program. We can, instead, test these parts with integration tests to be more effective. For instance, previously, we tested `Store` with a database. This enabled us to test its behavior as if it were in production. It's close to metal.

A database isn't an implementation detail but rather often an integral part of a program. It isn't easy to provide a test double that can verify the program's interactions with a database.

Note

Isolate business logic into pure functions to unit test them. On the other hand, integration tests are in between unit and end-to-end tests. They're high-level enough to test that the program works as expected and low-level enough to help us pinpoint the issues.

Avoiding premature abstractions

Unit testing `Store` isn't useful as it works directly with a database. What about handlers that indirectly work with a database through `Store`? If we want to test the handlers in isolation, we would need to use an interface to isolate `Store` from the handlers for the sake of testing. However, interfaces are most effective when multiple implementations exist.

Note

Avoid premature abstractions and interfaces without a clear reason for them to exist. Concrete types (i.e., `Store`) can manage implementation details on their own.

But `Store` doesn't have multiple implementations (test doubles don't count as they aren't needed in production). Using an interface to isolate it from handlers is unnecessary. It's better to use a real thing over a test double for realistic tests.

Writing integration tests for database interactions and unit tests for standalone business logic is more effective in my experience. Of course, you're free to choose your own approach.

11.5.1 Testing against the real thing

The "real thing" can be a database or any other software the program interacts with. By this, we don't mean testing against a staging or a production database (of course, it's good practice to test within these environments after we wrap our work and are ready to deploy).

We should test our code against the real thing as much as possible if it's viable. This approach increases the chances that code that works locally will also work in production.

For the link server, we use SQLite to run our tests against it. Our tests run blazingly fast and test against the real thing (we would use the same approach even if we used a more advanced database system such as PostgreSQL). Our program will most likely work in the production environment, too. In this section, we'll follow the same approach. Then, in the next section, we'll look at how to test the handlers using interfaces and test doubles.

Testing with a database

In the next listing, we're testing the `Shorten` handler against the database (the real thing the code uses). There are no test doubles in sight.

Listing 11.18 Testing with a database (link/server_test.go)

```
func TestShorten(t *testing.T) {

    t.Parallel()

    w := httptest.NewRecorder()
    r := httptest.NewRequest(http.MethodPost, "/", strings.NewRea
        `{"key": "go", "url": "https://go.dev"}`,
    ))

    store := NewStore(
        sqlxtest.Dial(context.Background(), t),
    )
    Shorten(store).ServeHTTP(w, r)

    if w.Code != http.StatusCreated {
        t.Errorf("got status code = %d, want %d", w.Code, http.St
    }
    if got := w.Body.String(); !strings.Contains(got, `"go"`) {
        t.Errorf("got body = %s\twant contains %s", got, `"go"`)
    }
}
```

Recall from the previous chapters that `NewRecorder` returns `ResponseRecorder` which allows us to observe what a handler responds (what the handler writes to `ResponseWriter`). We're also setting up a test request to send a POST request to the handler with a JSON payload.

Once we set up the recorder and the request, we connect to the database using `Dial` and pass the connection pool to a new `Store`. After that, we run `Shorten` with `Store`. Lastly, we check if the handler returns a correct HTTP status code and response.

Note

`strings.NewReader` returns an `io.Reader` that reads from a string. It's useful to pass a string to a function (or a method) that expects an `io.Reader` to do its work (i.e., `NewRequest`).

Running the test against the database

We've written a test that connects to the real database that our program uses to test the `Shorten` handler. Let's run the test ten thousand times, for example.

```
$ go test ./link -run=TestShort -count=10000
ok      github.com/inancgumus/gobyexample/bite/link    1.042s
```

Running ten thousand tests takes around a second. This includes the time running the handler, sending the query to the database, persisting the link, getting back a response, and so on. Of course, it might take longer for more complex SQL queries and program logic.

I've also measured the server with the HIT tool and discovered that the link server can handle fifty thousand requests per second (RPS). I've also changed the database from SQLite to PostgreSQL (without tuning it for performance) and ran twenty thousand RPS.

Note

For context, I'm running a MacBook Pro M1 with 10 CPU cores and 32 GB of memory.

Testing internal errors

We can also test for edge cases even if we test against the real database. For instance, we can disconnect from the database right before calling `Shorten` in a test to see how the handler would react to an internal error from the storage component.

See the next listing for an example.

Listing 11.19 Testing with a database (link/server_test.go)

```
func TestShortenInternalError(t *testing.T) {

    ...

    store := NewStore(
        sqlxtest.Dial(context.Background(), t),
    )
    _ = store.db.Close()

    Shorten(store).ServeHTTP(w, r)

    if w.Code != http.StatusInternalServerError {  #A
        t.Errorf("got status code = %d, want %d", w.Code, http.St
    }  #A
}
```

We've disconnected the store from the database to force the handler to respond with an internal error. Let's run this test.

```
$ go test ./link -run=TestShortenInternalError -v
ERROR internal url=/ message="creating link: sql: database is clo
--- PASS: TestShortenInternalError (0.00s)
```

The test passed because the handler responded with an internal server error. However, we see cluttering in the test logs because of the logger. This message is coming from the `httpio` package's `Error` function. Recall that it logs an error when an internal error occurs.

```
package httpio

func Error(err error) Handler {
    // ...
    return func(w http.ResponseWriter, r *http.Request) Handler {
        if code == http.StatusInternalServerError {
            slog.Log(r.Context(), slog.LevelError, "internal",
                "url", r.URL, "message", err)
            err = bite.ErrInternal
        }
        return Code(code, Text(err.Error()))
   }
}
```

Let's look at a solution.

TestMain

We can silence the logger output in `TestMain`.

`TestMain` is a special function that is useful for doing setup and teardown work:

```
func TestMain(m *testing.M) {

    // run setup work here
    m.Run()                #A
    // run teardown work here
}
```

The `testing` package first runs `TestMain` (if it exists) before any tests in the same package. The first parameter `testing.M` has a `Run` method that runs all the tests (and benchmarks) in the package. We need to call this method ourselves to initiate the test run.

Note

If `TestMain` is present in a package and `Run` is not called, no tests are run.

As the next listing shows, inside `TestMain`, we're setting the default logger level to the maximum possible integer to silence the logger throughout our program and tests. The `slog` package's logger won't log anything below this level (that is nothing will be logged).

Listing 11.20 Silencing the logger (link/link_test.go)

```
package link

import (
    "log/slog"
    "math"
    "testing"
)

func TestMain(m *testing.M) {
    // silence the logger
    slog.SetLogLoggerLevel(math.MaxInt)    #A
    m.Run()                #B
}
```

We've added `TestMain` to the `link` package to silence the logger. Let's give it a try.

```
$ go test ./link -run=TestShortenInternalError -v
--- PASS: TestShortenInternalError (0.00s)
```

No more internal error output in the logs for the tests.

Note

We could pass the logger as a dependency to the handlers, and we wouldn't need to silence the logger from TestMain. However, we would need to pass the logger in each test. For smaller programs, it's fine to use the same logger. For non-trivial programs, it's better to inject the logger. There are no hard and fast rules. Choose the best approach that makes sense.

Wrap up

Now that we looked at testing against the database, let's answer some possible questions.

One might wonder if switching to a different database would be difficult without an interface. We've already switched from an in-memory database to SQLite. We only had to change `Store`. This was possible because the types in our program have clear responsibilities.

Tip

Abstraction doesn't always mean declaring an interface to decouple types. Concrete types also have interfaces (e.g., Store's interface is its methods). Once we provide enough isolation between types, our programs are often more maintainable and flexible.

What about tests that might yield indeterministic results due to the database? In practice, this seldom happens. Moreover, setting up a deterministic database in a Docker container is quite straightforward. Since containers provide a controlled environment unaffected by external systems (when configured correctly), the likelihood of indeterminism drops significantly. Essentially, containers ensure the database environment is stable and consistent, minimizing the factors that could lead to unpredictable test outcomes.

Tip

Use Docker test containers to ensure consistent and isolated testing environments, reducing indeterminism in tests. To learn more, visit: golang.testcontainers.org

Encountering indeterminism can catalyze the identification of potential enhancements within our programs. By proactively addressing these inconsistencies, we prepare ourselves to manage such situations more effectively and contribute to developing more authentic tests.

To wrap up, testing against a real database isn't something to fear. If we hit some limits or our program is complex (relative), there are many ways to tune the database.

11.5.2 Testing with interfaces

Although testing against a database is advantageous, sometimes we can't involve it in our tests. It might be because of our company policy or because the underlying infrastructure is too complicated to set up properly in tests. In those cases, we can fake the database with *test doubles* using interfaces.

Note

The source code for this section is available in the book's repository at in the files bite/link/server.interfaces.go and bite/link/server.interfaces_test.go

Restructuring the server

Once we switch to an interface, there's no straightforward way to go back, so we should be careful before deciding. Still, we can reduce the impact by using unexported interfaces and functions. Let's recall the server's functions before testing it with an interface.

```
func Shorten(links *Store) httpio.Handler
```

```
func Resolve(links *Store) httpio.Handler
```

Each takes a concrete `Store`. What they need is explicit and easy to follow.

Imagine, over time, we added many methods to `Store` besides `Create` and `Retrieve`.

```
func (s *Store) Create(...)   { ... } // <- we only need this
func (s *Store) Retrieve(...) { ... } // <- we only need this
func (s *Store) Stats(...)    { ... }
func (s *Store) Search(...)   { ... }
// more methods...
```

Instead of making a 1:1 mapping to `Store`, we'll declare two interfaces only with the `Create` and `Retrieve` methods since we *only need* these two methods. This is our first line of defense to reduce the impact of premature abstraction the interface would introduce.

```
type linkCreator interface {

    Create(ctx context.Context, link Link) error
}

type linkRetriever interface {
    Retrieve(ctx context.Context, key string) (Link, error)
}
```

These interfaces declare what the handlers need: `Create` and `Retrieve`. If we add more methods to `Store`, we won't have to change the interfaces to adopt the new methods.

When defining an interface, keep it small with only the necessary methods. This approach leads to more modular and maintainable packages, easier testing, implementation swapping, and code reuse, making the code more concise and reducing unnecessary complexity.

Tip

Avoid large interfaces and declare interfaces only with a small set of methods needed.

Moreover, if we exported the interfaces (e.g., `linkCreator`), other packages could depend on them. Any modifications (e.g., adding new methods) we might make to the interfaces would break those packages. We avoid that problem by declaring unexported interfaces.

Unexporting the interface was our second line of defense. These unexported interfaces are only for testing purposes and won't show up in the package documentation.

Tip

Avoid exporting interfaces declared for testing purposes to reduce dependency issues.

Restructuring to unexported interfaces

Our interfaces are unexported. So, unexported functions should taken them as input.

One of the common mistakes Go beginners make is to declare unexported interfaces as an input parameter for an exported function. This might confuse package users as unexported interfaces and their methods won't show up in the package documentation.

Tip

Avoid unexported parameter types in exported functions and methods.

Instead, exported handler functions will continue to take `Store`, but inside, they'll forward the call to unexported functions that take interfaces as input and do the real work.

```
Shorten(*Store) -> shorten(linkCreator)
Resolve(*Store) -> resolve(linkRetriever)
```

Each exported server function will continue to take `Store`, but inside, they forward the call to an unexported function that does the actual work and takes the interfaces.

```
func shorten(links linkCreator) httpio.Handler {

    /* original code */
}

func resolve(links linkRetriever) httpio.Handler {
    /* original code */
}
```

Now that we've unexported the original functions, let's declare exported functions that forward calls to these unexported functions. We do so to avoid exporting the interfaces.

```
func Shorten(links *Store) httpio.Handler { return shorten(links)

func Resolve(links *Store) httpio.Handler { return resolve(links)
```

Our last line of defense was to take the interfaces only from unexported functions.

Reviewing the final code

This is what the final code looks like:

```
func NewServer(links *Store) *Server      { ... }
func Shorten(links *Store) httpio.Handler { return shorten(links)
func Resolve(links *Store) httpio.Handler { return resolve(links)

type linkCreator interface {
    Create(ctx context.Context, link Link) error
}

func shorten(links linkCreator) httpio.Handler { ... }

type linkRetriever interface {
    Retrieve(ctx context.Context, key string) (Link, error)
}

func resolve(links linkRetriever) httpio.Handler { ... }
```

We've restructured the handlers without changing the package's usage. On the outside, the handler functions are still the same. But inside, they can now work with any type with a `Create` and `Retrieve` method. This enables testing them with a test double.

The existing tests will also continue to work as `Store` satisfies these interfaces.

```
$ go test ./link/...
PASS
```

This "./..." syntax runs all tests in the link folder and subdirectories.

Crafting a test double

Now that the handlers take interfaces, we can provide a test double. As an example, let's provide a function that will allow us to convert a function to `linkCreator`:

```
// inside the test code:

type fakeLinkCreator func(context.Context, Link) error

func (f fakeLinkCreator) Create(ctx context.Context, link Link) e
    return f(ctx, link)
}
```

Note

This function type is an adaptor and it's similar to the `HandlerFunc` type we saw.

The `fakeLinkCreator` type implements the `Create` method and hence satisfies the `linkCreator` interface. We can use it to convert ordinary functions to test doubles:

```
fake := func(context.Context, Link) error {    #A
    return nil
}
creator := fakeLinkCreator(fake)            #B
handler := shorten(creator)            #C
```

We convert the `fake` closure to `fakeLinkCreator` that satisfies `linkCreator`. The closure simulates creating a link and returns a `nil` error to signify the link was successfully created. Then, we pass `creator` (which we use as a test double) to the `shorten` handler. This is possible because shorten expects `linkCreator` and `creator` satisfies `linkCreator`.

Testing with the test double

Now that we have a test double, we can write a test. See the next snippet below.

```
func TestShorten(t *testing.T) {
    creator := fakeLinkCreator(func(context.Context, Link) error
        return nil
    })
    handler := shorten(creator)

    body := `{"key": "go", "url": "https://go.dev"}`
    w := httptest.NewRecorder()
    r := httptest.NewRequest(http.MethodPost, "/", strings.NewRea
    handler.ServeHTTP(w, r)

    if want := http.StatusCreated; w.Code != want {
        t.Errorf("got status code = %d, want %d", w.Code, want)
    }
}
```

Once we run the test, it'll pass because the `shorten` handler doesn't receive an error from the fake creator. We no longer need to connect to a database for testing.

Exercise

Write a test double adaptor (similar to `fakeLinkCreator`) for the retrieve handler.

Exercise

Test the `retrieve` handler with a test double.

Exercise

Test the `shorten` and `retrieve` handlers for edge cases.

Wrap up

Interfaces, while useful, have downsides.

Firstly, they can make code navigation and understandability harder.

When we encounter a function that takes an interface in our code, it can be difficult to figure out how to proceed. We must dig into what concrete types we pass to the function to understand how our code works. This process can become cumbersome, especially when we're dealing with many interfaces, which tend to clutter our codebase, making it harder to understand. For instance, what does the following function need to work?

```
func shorten(links *Store) httpio.Handler
```

As it takes a concrete type, it's explicit and straightforward to see that it needs `Store`.

What about the following function? Which concrete type do we pass to this function?

```
func shorten(links linkCreator) httpio.Handler
```

The answer is any type with a `Create` method. What this function takes as an input is implicit. We must follow the trail in our codebase and find out which concrete type we pass to this function to understand the code flow (especially crucial while debugging or confronted with an unfamiliar code base that we need to add a new feature). One might think what concrete type this function takes doesn't matter because it takes an interface. Or this might seem trivial because modern text editors can help, but context switching during development can lead to mistakes and cognitive overload even though we have perfectly designed code.

Secondly, we have to maintain test doubles when we use interfaces.

Often, we rely on test doubles to simulate the behavior of our real types, and this extra layer of abstraction means we're juggling more components, which increases the maintenance workload. Keeping the test doubles in sync with our evolving codebase adds complexity.

Lastly, using interfaces can lead us to overlook realistic testing scenarios.

We may focus too much on the abstract rather than the concrete, leading to tests that pass with flying colors but fail in production where database usage is crucial. We must balance our testing strategies to ensure they reflect the actual use cases our code will encounter.

Let's wrap up:

- Interfaces can make code navigation and understanding difficult.
- Test doubles are another piece of code that we have to maintain. And they can lead to unrealistic tests that don't match the behavior we expect in production.

11.5.3 Wrap up

Interfaces can be useful for decoupling, but their misuse can lead to unnecessary complexity. The best code is easy to write, read, and maintain. Avoid using interfaces solely for testing purposes and only add them when there is a genuine need for extensibility. Every interface should earn its place in the codebase. Keeping things concrete and prioritizing real implementations over test doubles can make maintenance and understanding easier.

Note

As Rob Pike says: "Don't design with interfaces, discover them."

Go's structural type system and implicit interfaces let us start with concrete types and discover interfaces later. So, a better solution often starts with a concrete type, and then lets the consumer packages decide whether to declare a small interface for their own needs. This approach makes the code more concise and reduces unnecessary complexity.

We'll dive deeper into what this means in the next section.

11.6 Designing idiomatic interfaces

Before wrapping up this chapter, let's talk about designing idiomatic interfaces.

We'll discuss two topics:

- Where should we declare interfaces?
- How to design better interfaces?

Let's dive in.

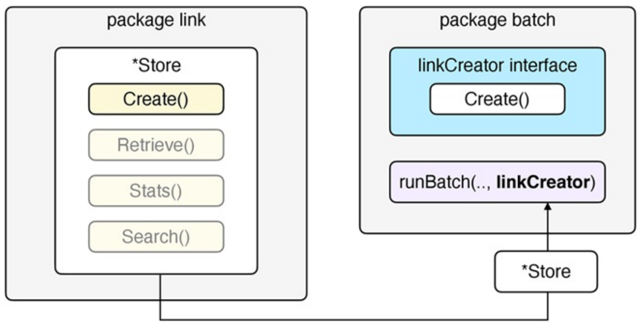
11.6.1 Where should we declare interfaces?

The idiomatic approach is to let a package's consumers define the interfaces they need.

This is possible because interfaces in Go are implicit. For instance, rather than the `link` package providing an interface for its consumers, consumers can define their own interfaces to abstract `Store`'s methods that they're interested in. Let's look at an example.

Say there's another package called `batch` that wants to create links. See Figure 11.8.

Figure 11.8 The interface in the batch package defines only what runBatch needs. We can pass a Store to batchRun because Store satisfies linkCreator.



This package fetches links from another database or system and uses `Store` to create links. However, instead of directly depending on `Store`, it declares *its own interface* to decouple from `Store`. The code looks like this:

```
package batch

type linkCreator interface {
    Create(ctx context.Context, link hit.Link) error
}

func runBatch(ctx context.Context, lc linkCreator) error {
    // creates prepared links from another DB
    // ...
    err := lc.Create(ctx, hit.Link{...})    #A
    // ...
}
```

So, we can pass `Store` or any other type with a `Create` method to `runBatch`:

```
package batch

func Run(..) error {
    // ...
    err := runBatch(ctx, link.NewStore(...))
    // ...
}
```

Keep in mind that this (declaring interfaces on the consumer side) isn't a "rule" written in stone but rather a suggestion. There are many packages in the standard library that contain both interfaces and the concrete types that satisfy them.

For instance, take the `FileSystem` type declared in the `net/http` package.

```
package http

type FileSystem interface {
    Open(name string) (File, error)
}
```

One implementation of this interface is in the same `net/http` package:

```
package http

type Dir string

// Open implements FileSystem using os.Open,
// opening files for reading rooted and relative
// to the directory d.
func (d Dir) Open(name string) (File, error)
```

This interface is exported because the `net/http` package wants others to implement it. Moreover, the users of this interface are in the same `net/http` package:

```
package http

func FileServer(root FileSystem) Handler
func NewFileTransport(fs FileSystem) RoundTripper
```

To summarize, we should choose the most convenient route for our situation. Suggestions are good, but they are general pieces of advice. We have the freedom not to follow them.

Tip

It's not necessary to worry about interfaces that have the same names and methods, such as `linkCreator` in the `link` and `batch` packages. In Go, the important factor is the methods themselves. The language has a structural type system, which means that interface names don't have a significant impact. For instance, we could have an interface called `store` (instead of `linkCreator`) with a `Create` method and pass `Store` to `runBatch`. Named interfaces are helpful for documentation purposes or when composing other interfaces. Take, for example, `io.ReadCloser`, which is a combination of `io.Reader` and `io.Closer` interfaces.

11.6.2 How to design better interfaces?

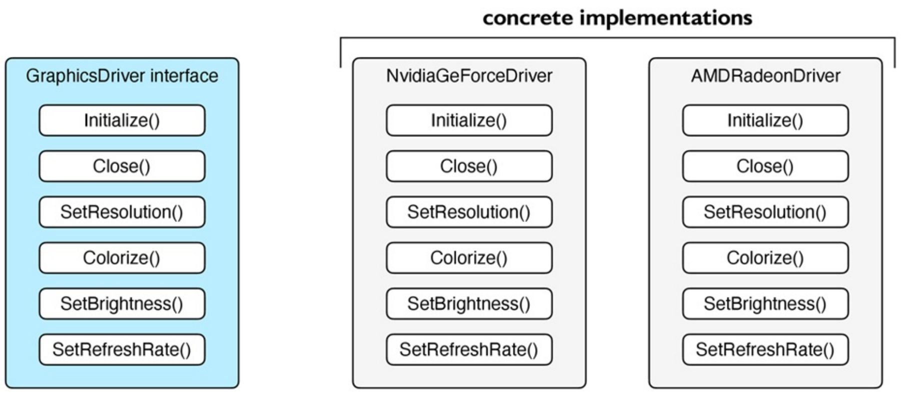
Avoid large interfaces that demand too much from concrete types. Define smaller, specific interfaces with only necessary methods to make it easier for concrete types to satisfy them, and to make packages more modular, maintainable, and reusable.

When designing interfaces, it's more useful to focus on the behavior or operations that types should perform, rather than their implementation details or data types. This means that instead of broad interfaces for general categories of types, it's better to declare more specific interfaces that define particular behaviors or operations that the types should support.

For example, we might want to declare an interface for graphic device drivers. A graphic driver is software that allows the operating system to work with the graphics hardware and manage display output, such as a driver for Nvidia GeForce, AMD Radeon, Intel, and so on.

See Figure 11.9 for an illustration of this interface and its implementations.

Figure 11.9 The 1:1 mapping between the GraphicsDriver interface and concrete types makes it necessary for all implementers to update their code every time there is a change in the interface methods. This can be a cumbersome task.



The `GraphicsDriver` interface groups concrete graphics driver types *by category*. It has a 1:1 mapping to the concrete types. This limits which concrete types can implement the interface (only the concrete graphics driver types), reducing the interface's reusability.

Let's take a look at this interface in the code:

```
type GraphicsDriver interface {

    Initialize() error
    Close() error
    SetResolution(width, height int)
    Colorize(r, g, b uint8)
    SetBrightness(level int)
    SetRefreshRate(rate int)
}
```

This interface is designed to correspond to all the methods of the concrete types. Each method defined in the interface is also present in the concrete types. The interface and the concrete types force a type hierarchy, which isn't how interfaces in Go are meant to be used.

Note

Go doesn't allow type hierarchies. Instead, we declare interfaces to group types by behavior rather than types (unlike classical object-oriented programming languages).

Moreover, this close mapping limits the interface's flexibility and reusability since only these concrete graphics driver types with all these methods can satisfy it. Suppose another type of driver has a different set of functionalities. In that case, it cannot be used interchangeably with this interface, which reduces the interface's potential for reusability in different contexts.

Instead of declaring `GraphicsDriver` to group concrete types by category, it's more effective to declare generic interfaces for the *behaviors* we expect. For instance:

```
type RGBColorizer interface {

    Colorize(r, g, b uint8)
}

type BrightnessSetter interface {
    SetBrightness(level int)
}

type RefreshRateSetter interface {
    SetRefreshRate(rate int)
}
```

These interfaces specify behavior rather than a type category or hierarchy. Unlike `GraphicsDriver`, they're focused and more effective. Concrete implementations don't need to fit into a rigid category, enabling diverse and flexible implementations. For example, an image processing type that manipulates the colors of individual pixels might satisfy `RGBColorizer`, and another type that controls the brightness of a set of lights can satisfy `BrightnessSetter`. Or a type in a video streaming server can satisfy `RefreshRateSetter`.

Because these focused interfaces are declared for abstracting behavior rather than planning for the concrete types that will satisfy them, their implementations will be various and vast.

Effective interfaces often lack knowledge of the types that will implement them, and their implementations can reach far beyond the concrete types that initially implement them.

Tip

Avoid trying to fit interfaces into type hierarchies but focus on general behavior.

11.6.3 Wrap up

Declaring an abstract enough interface is an art in itself, and it depends on the situation. So, we'll wrap up the section with some ideal principles, but of course, not rigid rules.

- Declare interfaces where they're used, not where they're implemented.
- Prefer small interfaces over broad ones for reusability and composability.
- Interfaces should focus on particular behaviors rather than categories.
- Interfaces should be abstract enough that unknown future types can satisfy them.

11.7 Exercises

1. Experiment with different SQL drivers (e.g., for MySQL) and modify the code accordingly. Research available SQL drivers and modify the connection setup.
2. Create a separate timeout context (`context.WithTimeout`) in `linkd` while connecting to the database. This ensures this operation won't wait forever.
3. Introduce data expiration for shortened links by allowing them to expire after a specified period. Modify `Store` to store expiration timestamps and implement a cleanup process to remove expired links from the database periodically. Consider launching a goroutine before launching the server to handle the cleanup process.
4. Implement data compression for stored links using `Valuer` and `Scanner` interfaces. Employ data compression techniques to reduce storage space requirements for links in the database. Update the storage interface and

`Store` implementation to handle link data compression and decompression. Test the compression feature to verify its effectiveness and storage space savings.
5. Incorporate a `Stats` service into the server, which could be part of the `link` package. This service would provide click statistics for shortened links.
6. Add your own methods to the `sqlx.DB` type and log each time `ExecContext` and `QueryContext` methods are called.

11.8 Summary

- Isolation from infrastructure components enhances maintainability. The `sqlx` package, for instance, separates database operations from the rest of the code.
- Go's `sql` package allows working with any SQL database using third-party drivers.
- Database drivers are typically registered via side-effect importing, invoking their `init` functions. To enhance maintainability, avoid `init` functions.
- `DB.Open` probes the driver registry, returning a handle to a connection pool.
- `sql.DB` provides numerous settings to optimize the connection pool's performance, such as `SetMaxOpenConns` and `SetMaxIdleConns`.
- `sql.DB` should be preserved throughout the lifespan of a program.
- `DB.Dial` yields a connection pool initially devoid of connections.
- `DB.PingContext` establishes a connection to the database and verifies its authenticity.
- `DB.ExecContext` retrieves a connection from the pool, executes the query, and returns the connection to the pool.
- `DB.QueryRowContext` retrieves a single record from the database. `DB.Scan` injects database data into a Go type and relinquishes the connection to the pool. Not coupling `QueryRowContext` with `Scan` may lead to database connection leaks.
- Placeholders offer value substitution in SQL queries and mitigate injection attacks.
- The `sql.Scanner` interface permits modifications to a value post retrieval from a database column, while `driver.Valuer` functions in the

reverse manner.

- The embed directive allows file(s) to be incorporated into the final binary.
- `errors.As` assigns the error to a designated error variable, returning true if the error type is within the error chain.
- Using `testing.TB` instead of `*testing.T` permits a test helper to be used in tests and benchmarks.
- Testing against real infrastructure as opposed to test doubles gives an accurate measure of actual behavior. Go's structural type system and implicit interfaces enable starting with concrete types and subsequently discovering interfaces.
- Design interfaces around the behavior types should support, not their category.
- Smaller, focused interfaces with requisite methods are easier for concrete types to satisfy. Interfaces better be declared where they're used. Return concrete types, and let consumer packages declare interfaces.